From Scenarios to Optimally Allocated Timed Automata

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Sandeep Vuppula

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Dr. Neda Saeedloei

June 2017

Acknowledgements

I would first like to thank my advisor Dr. Neda Saeedloei for her constant guidance and support in completing my thesis. Without her motivation and careful supervision, this work would never have taken shape.

I would like to thank my committee members, Dr. Haiyang Wang, and Dr. Ping Zhao for their time and valuable comments to better my work. I would like to thank my graduate instructors Dr. Peter Peterson, Dr. Neda Saeedloei, Dr. Ted Pedersen, and Dr. Richard Maclin for sharing their vast knowledge and helping me understand the theoretical and practical concepts in Computer Science. I would also like to thank Lori Lucia, Clare Ford and Jim Luttinen for their assistance.

I would like to thank the members of my research group, Swathi Vallabhajosyula and Vaclav Hasenohrl for their ideas and discussions during the completion of my thesis.

Finally, I would like to thank my parents and my sister for their unconditional love and trust during all times of my life. Without them, none of my success would have been achieved.

Dedication

I dedicate this thesis to

my mom,

Mrs. Praveena Devi Vuppula,

my dad,

Mr. Ramachandram Vuppula,

my sister,

Ms. Swetha Vuppula,

and all of my friends.

Abstract

Our contribution is twofold. First, we develop a new method for synthesizing a formal model for real-time systems from scenarios. Scenarios describe partial behaviours of real-time systems during some time interval. We propose Timed Event Sequences to formally express scenarios. Given a set of scenarios as Timed Event Sequences, along with a mode graph, we propose a method to automatically construct a *minimal*, *acyclic*, and *deterministic* timed automaton that models the specified aspects of the system.

Second, we consider the problem of optimally allocating clocks in timed automata. Reducing the number of clocks in a timed automaton is important, as it directly affects the complexity of the verification problem. Given a timed automaton, it is *undecidable*, in general, to check whether there exists another timed automaton that accepts the same language but has fewer number of clocks [9]. We identify a fairly general class of timed automata and propose an algorithm (polynomial time) for optimally allocating clocks to timed automata in this class. The previous approaches [10, 6], changed the shape of the graph of the original timed automaton by constructing bisimilar timed automata. Our method does not change the graph of timed automata. The cost of our algorithm is quadratic in the size of the underlying graph of the timed automaton.

Contents

List of Figures

1 Introduction

Model-based design is a very effective method for designing real-time systems. Building a formal model is very useful in the process of analysis, design and verification of complex systems. A formal model of a system will give us a better realization of how the final implementation of the system will behave in real world. However, building formal models for systems is challenging because of the lack of good formal requirements specifications. The requirements are often given in low level or they are incomplete and ambiguous. Low level requirements are hard to understand and cannot be used for modeling. Moreover, incomplete and ambiguous requirements often lead to unwanted behaviours. In most real-time systems, where safety is critical, such unwanted behaviours are not acceptable. Modeling a system formally can help us to understand the desired and undesired behaviours of the system. Thus, modeling is crucial in the process of constructing real-time systems due to the lack of satisfactory formal requirements. Before building the model of a system, the following questions should be answered first:

1. How the requirements should be expressed formally, and

2. How the formal model of a real-time system can be constructed from requirements.

One approach for building a formal model is using scenarios. A scenario is a partial description of the behaviour of a system. Building a formal model from scenarios will help us to build a model of the system and understand the intricacies involved in developing such systems. The formal model that we will build is timed automata [2]. Our method for constructing such automata includes two steps:

1. First, we synthesize a timed automaton from a set of scenarios.

2. Second, we optimally allocate clocks in the constructed timed automaton.

Scenarios should be expressed formally for constructing a formal model of a real-time system. We introduce *Timed Event Sequences (TES)* to represent scenarios formally. We use *mode graphs* to specify the legal events that can occur in the system. The mode graph depicts the interaction between various modes of a system. A set of scenarios represented as Timed Event Sequences (TES) and a mode graph are used to generate a deterministic, acyclic timed automaton, with minimum number of states.

Most of the work for building a formal model from requirements has been done in the context of non-timed systems. Not much work has been done on synthesizing formal models of real-time systems from scenarios [17, 16, 5]. The closest work to our method is the work of *Somé et al.* [15]. The proposed algorithm introduces non-determinism to the constructed automaton. Moreover, many concepts such as "super states" (sup-states) and "characteristic conditions" are not formally defined. Allocating clocks and generating clock constraints and clock resets are not described formally. More importantly, the number of clocks in the constructed automaton is not optimal.

On the contrary, our work clearly defines a set of criteria for a set of scenarios in order to make the construction of a minimal, acyclic and deterministic timed automaton from a set of scenarios feasible. Moreover, we propose a method for optimally allocating clocks in the constructed timed automaton. Our timed automaton belongs to a class of timed automata that satisfies the following properties:

- A clock t_j can be reset only on the transitions emanating from a state labelled j and,

- A clock in a clock constraint on a transition r from a state q can only refer to a clock that has been reset on a transition leaving a state that dominates q. We call this *dominance assumption*.

It is well known that, the number of clocks in a given timed automaton has a direct impact on verification of the system. The more number of clocks, the harder it is to verify the system modeled as a timed automaton. Moreover, as the number of clocks increases, the complexity of the system increases and sometimes, it is impossible to verify a system without approximating methods.

Given a timed automaton $\mathcal{A}$, the problem of deciding whether there exists another timed automaton $\mathcal{B}$ that accepts the same language as that of $\mathcal{A}$ but with fewer number of clocks is *undecidable* [9]. However, for our class of timed automata, we propose an optimal clock allocation algorithm whose complexity is quadratic.

Given a timed automaton that belongs to our class, we use liveness analysis of clocks to determine the minimum number of clocks, and then use this results to optimally allocate clocks. Liveness analysis of clocks helps us to reuse the clocks: if two clocks have disjoint liveness ranges, we can substitute those two clocks by only one clock.

To show the effectiveness of our approach, we apply our synthesis and optimal clock allocation methods to some real-life examples.

2 Background

As part of the background chapter, timed automata is presented with syntax and semantics. Then, a brief introduction of temporal logics, which is used for specifying and verifying the properties of concurrent systems is given. We also present a brief introduction of UPPAAL, a model checker for timed automata.

2.1 Background

A real-time system takes input from its surrounding environment and produces results within a stipulated amount of time. The temperature controller, anti lock breaking system and aircraft controller are some of the examples of real-time systems. The correctness of such systems depend not only on the correctness of the output but also on the satisfiability of timing constraints. For example the response of a program must be generated within a predefined amount of time. Thus, most of the systems used for the mission critical applications are real-time systems, where the time at which the response is produced should be within a certain timeline, else the system is said to have failed.

In the real world, the behaviour of almost every system changes according to time. For example, in a reactive system such as a controller, time is the most important factor to consider while modeling the behaviour of the controller. To model such real-time systems, Rajeev Alur and David L.Dill introduced the theory of timed automata by extending the finite state automata with clock variables [2]. Before describing timed automata, we present a brief overview of different approaches for modeling time, finite state automata and ω-

automata.

2.1.1 Modeling Time

There are three approaches for modeling time [2].

Discrete time model: It considers time to be a monotonically increasing sequence of integers. That is, time is viewed as a discrete variable. For example, in a digital clock each tick can be associated with an integer value whose value increases by one at every step. This kind of model can be easily transformed into formal language by inserting a silent event between the events so that the time of occurrence of each event will be the same as its position, there by discarding the time sequence. But this in turn limits the preciseness with which real-time systems can be modeled, where, all the events do not occur at integer times.

Fictitious-clock model: It is similar to that of discrete time model except that it assumes sequence of times to be non decreasing integers. Even if the events occur at real valued clock times, only the integer values with respect to a digital clock are considered in the time trace. These models can be transformed into formal languages by representing time in an approximate sense. But, this limits accuracy in modeling the physical systems as exact time values at which the events occur are not considered.

Dense time model: In this model, the domain of time is considered as a dense set and the time of occurrences of events as real numbers, which increase monotonically without any limit. Due to the difficulties in transforming dense time traces into formal languages, timed automata was developed for analysing such systems.

2.1.2 Finite State Automata

A finite state automaton (FSA) or a finite state machine (FSM) is an abstract machine which has a finite number of states. On an input, the machine changes from one state to another state: this is called a transition.

Definition (Finite State Automata): A finite state automaton is a 5 tuple $(Q, \Sigma, q_0, F, \delta)$ where,

- Q is a set of states,
- Σ is a finite set of input alphabets,
- $q_0 \subseteq Q$ is a set of start states,
- F is a set of final states,
- $\delta : Q \times \Sigma \rightarrow Q$ is a transition function.

The set of final states F can be an emptyset (ϕ), which implies that, the automaton does not have any accepting states.

2.1.3 ω-automata

An ω-automaton is elucidated as follows. An ω-automaton accepts infinite words. The language accepted by an ω-automaton is called an ω- language. An ω- language over a finite alphabet Σ is a subset of Σ^ω – the set of all infinite words over Σ.

Definition (ω-automata): An ω-automaton A is a tupple (S, S_0, Σ, E) where,

- S is a set of states,
- $S_0 \subseteq S$, is a set of start states,
- Σ is a finite set of input alphabets.

- $E \subseteq S \times S \times \Sigma$ is a set of edges,

If there is a state change from s to s_1 on input symbol a then it is represented as (s, s_1, a), i.e., $(s, s_1, a) \in E$.

A run r of A over $\sigma = \sigma_1\sigma_2\sigma_3...$ where $\sigma_i \in \Sigma, i \geq 1$ is

$$r \ : \ s_0 \xrightarrow{\sigma_1} s_1 \xrightarrow{\sigma_2} s_2 \xrightarrow{\sigma_3} ...$$

where $s_0 \in S_0$ and $(s_{i-1}, s_i, \sigma_i) \in E, \ \forall \ i \geq 1$.

The set $inf(r)$ for such a run r consists of states s_i where $s_i \in S$ for infinitely many $i \geq 0$. Different notions of acceptance are defined for ω-automa. We only consider B$\ddot{u}$chi and Muller automaton [2].

B$\ddot{u}$chi Automaton: A B$\ddot{u}$chi automaton A accepts exactly those runs r for which $inf(r) \cap F$ is not empty where F is a set of accepting states, i.e. there is an accepting state that occurs infinitely often in r.

Muller Automaton: A Muller automaton A with an acceptance family $\mathcal{F} \subseteq 2^S$ accepts exactly those runs r for which $inf(r) \in \mathcal{F}$.

2.1.4 Timed Automata

In automata theory, a timed automaton is a finite state automaton extended with a finite set of real-valued clocks. During a run of a timed automaton, all clock values increase with the same speed. The transition tables used in timed automata are timed transition tables which read timed words. Upon a transition the selection of next state is based not only on the input symbol but also on the time of the current symbol with respect to the formerly read symbols. The time of the current symbol is stored in a clock variable and it is compared to a time constant. Moreover, the clock values can be reset along those transitions.

Definition (Timed Automata): A timed automaton $\mathcal{A}$ is a tuple (Σ, S, S_0, C, E) [2] where,

- Σ is a finite alphabet,

- S is a finite set of states,
- $S_0 \subseteq S$ is a set of start states,
- C is a finite set called the clocks of $\mathcal{A}$,
- $E \subseteq S \times S \times \Sigma \times 2^C \times \Phi(C)$ is a set of edges, called transitions of $\mathcal{A}$, where the set $\Phi(C)$ of boolean *clock constraints* δ is defined by,
  $\delta := c \ \leq \ x \mid c \ \geq \ x \mid c \ = \ x \mid \neg \ \delta \mid \delta_1 \ \wedge \ \delta_2$
  where c is a clock in C and x is a constant in the set of non-negative rationals $\mathbb{Q}$.

A *clock interpretation* v for a set C of clocks assigns a real value to each clock; that is, it is a mapping from C to $\mathbb{R}$.

An edge $(s, s', a, \lambda, \delta) \in E$ is a transition from state s to s' on input symbol a. $\lambda \subseteq C$ gives the clocks to be reset with this transition and δ is a clock constraint over C.

Timed Sequence is an infinite series of time values $\tau = \tau_1\tau_2....$ where $\tau_i \in \mathbb{R}$ with $\tau_i \geq 0$ satisfying:

1. Monotonicity: τ monotonically increases, i.e., $\tau_{i+1} > \tau_i, \ \forall i \geq 1$,
2. Progress: $\forall t \in \mathbb{R}$, there is some $i \geq 1$, such that $\tau_i \geq t$.

Timed words are infinite sequences in which each symbol is associated with a real valued time of occurrence. A timed word over an alphabet Σ is a pair (σ, τ) where $\sigma = \sigma_1, \sigma_2...$ is an infinite word over Σ and τ is a timed sequence.

A run r of $\mathcal{A}$ over a timed word (σ, τ) is of the form

$$r :< s_0, v_0 > \xrightarrow[\tau_1]{\sigma_1} < s_1, v_1 > \xrightarrow[\tau_2]{\sigma_2} < s_2, v_2 > \xrightarrow[\tau_3]{\sigma_3} ...$$

where $s_i \in S$ and $v_i \in [C \rightarrow \mathbb{R}^+], \ \forall i \geq 0$, satisfying the following conditions:

- Initiation condition: $s_0 \in S_0$ and $v_0(x) = 0, \ \forall x \in C$,

- Consecution condition: An edge of the form $(s_{i-1}, s_i, \sigma_i, \lambda_i, \delta_i)$ exists, such that $(v_{i-1} + \tau_i - \tau_{i-1})$ satisfies δ_i and v_i equals $[\lambda_i \mapsto 0](v_{i-1} + \tau_i - \tau_{i-1})$.

Example: Consider the alphabet set $\{x, y\}$. A timed language L_1 which consisits of all the timed words (σ, τ) such that, there is no y after 4 units of time is given by

$$L_1 = \{(\sigma, \tau) | \forall i, ((\tau_i > 4) \rightarrow (\sigma_i = x))\}$$

Example: Consider the timed automaton of Figure 2.1.

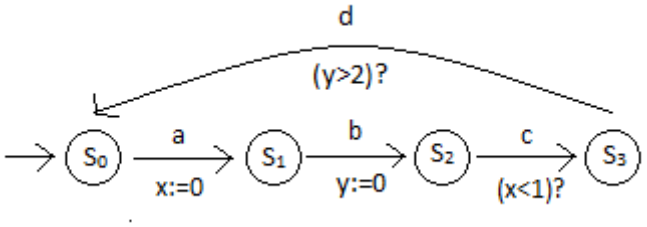


Figure 2.1: Timed automaton

$(a, 2), (b, 2.7), (c, 2.8), (d, 5), ...$ is a timed word accepted by this automaton.

An example of a run of the automaton over this timed word is given by:

$$< s_0, [0, 0] > \xrightarrow[2]{a} < s_1, [0, 2] > \xrightarrow[2.7]{b} < s_2, [0.7, 0] > \xrightarrow[2.8]{c} < s_3, [0.8, 0.1] > \xrightarrow[5]{d} < s_4, [3, 2.3] > ...$$

2.1.5 Model Checking

Model checking is a technique for automatically verifying concurrent systems having finite number of states [4]. The process of model checking involves three steps: `modeling`, `specification` and `verification`. Modeling is the process of transforming a given system design into a formal model that is acceptable by a model checking tool. Then the set of properties that the model should satisfy are specified in some temporal logic. In order to verify whether a given system satisfies the specification or not, the model checker performs an exhaustive search on the system's state space. If the system, does not satisfy a given specification, an error trace is generated by the tool to determine the cause of the error.

Temporal Logic:

Temporal logic is a formalism which is used for specifying the properties of concurrent systems, such as reactive systems which respond to external events [4]. For these systems, the correctness of the system depends not only on the input or output but also on the times during which that input or output occurs. Temporal logics are very useful because, they help us describe the ordering of events according to time without actually introducing the time explicitly into the system. Based on the assumption whether the time is linear or branching, temporal logic can be classified as two types *Linear Temporal Logic* (LTL) and *Computation Tree Logic* (CTL).

Kripke Structure:

A Kripke Structure is a labelled state-transition, where each node is mapped to a set of properties that should hold in that state. An atomic proposition is a statement that is true or false in a state. Assume AP is a set of atomic propositions.

Definition (Kripke Structure): A Kripke structure is a 4-tuple, $M = (S, I, R, L)$, where

- S is a finite set of states,
- $I \subseteq S$ is a set of initial states,
- $R \subseteq S \times S$ is a transition relation, where $(s, s') \in R$, if there is a transition from s to s' in M.
- L is a labelling function, from the set of states to the power set of atomic propositions (AP), i.e., $L : S \rightarrow 2^{AP}$.

Temporal operators:

In a labelled state transition system (e.g., a Kripke structure), a computation tree is used to represent the paths starting from each of the initial states. In temporal logics we discuss here, we do not mention time explicitly. Instead we use a formula, that specifies that

eventually some state is reached or some error state is never reached. Temporal operators are the special operators used to describe the properties like *eventually* or *never* [4]. The temporal operators are as follows.

- **X** (Ne**X**t time) specifies that a property holds in the next state of the path,
- **F** (**F**uture) specifies that a property eventually holds at some future state in the path,
- **G** (**G**lobally) specifies that a property holds at each and every state in the path,
- **U** (**U**ntil) specifies that the first property holds on every preceeding state in the path until the second property is true at some state in the path,
- **R** (**R**elease) specifies that the second property holds at every state in the path up to and including the state at which the first property holds.

Future, **G**lobally, and **U**ntil can alternatively be denoted as follows:

F - ◊

G - □

U - ◯

The temporal operators ◊ and □ can be combined to form new temporal operators as follows:

- □ ◊ φ - infinitely often φ
- ◊ □ φ - eventually forever φ

Linear Temporal Logic:

In Linear Temopral Logic (LTL) time is considered to be linear. There may be infinite number of states but, at any point of time there will be a unique successor. Let a be an

atomic proposition, the syntax of LTL [1] is given by,

$$\varphi \ ::= \ a \mid (\varphi \ \wedge \ \varphi) \mid (\neg \varphi) \mid (\mathbf{X} \, \varphi) \mid (\varphi \, \mathbf{U} \, \varphi)$$

2.1.6 UPPAAL

UPPAAL is a popular model checking tool for timed automata [11]. It is widely being used for automatically verifying the properties of real-time systems. It is pertinent in areas where timing details are critical, for example controllers. UPPAAL is used to verify reachability, safety and bounded liveness properties by modeling the system as networks of timed automata. The tool is first of its kind developed for model checking of timed automata.

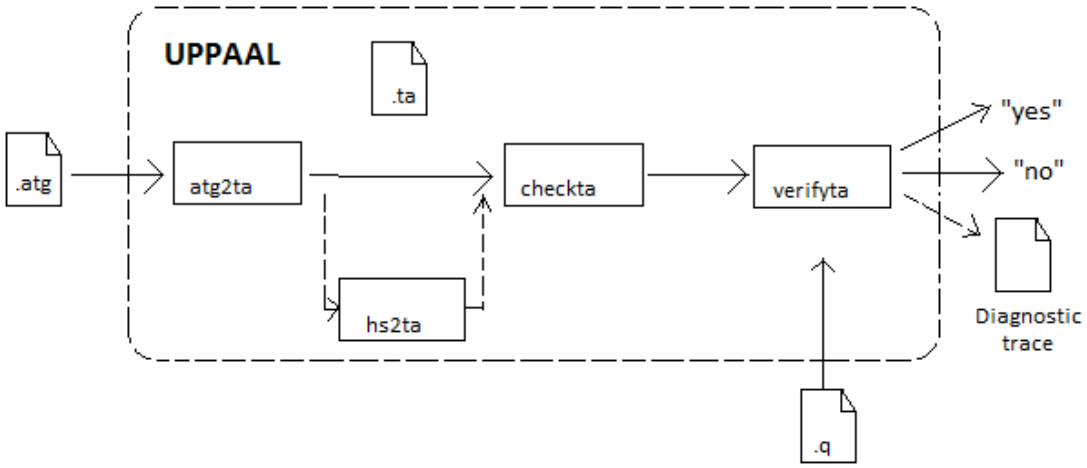


Figure 2.2: Overview of UPPAAL.

The system descriptions, represented as networks of timed automata can be defined using a textual (*.ta*) format or a graphical (*.atg*) format.The descriptions in graphical format are automatically compiled to a textual representation using *atg2ta* compiler. UPPAAL can also be used for analysing certain types of hybrid automata called linear hybrid automata. The linear hybrid automaton is converted to timed automata using *hs2ta* translator. *checkta* program performs syntax checking. It checks whether the clocks, variables and channels

are in harmony with the given declaration. The program *verifyta* performs model checking in UPPAAL. It also generates a diagnostic trace whenever the system fails in real time, making it easy for us to debug.

3 Synthesis of Timed Automata from Scenarios

Our contributions are as follows:

1. Synthesis of timed automata from a set of scenarios and
2. Optimally allocating clocks for the constructed timed automata

Our first contribution involves developing and implementing an algorithm for constructing a timed automaton from a set of scenarios. The second contribution involves an algorithm for reducing the number of clocks in the constructed automaton.

We use an invariant of an Automated Teller Machine (ATM) as an example to explain scenarios and our synthesis method. In its initial state, the ATM machine waits for a user to insert his bank card. Once the card is inserted, the user can either cancel the transaction within 4 seconds or enter his PIN within 5 to 60 seconds. If the transaction is cancelled, the ATM ejects the card within 2 to 3 seconds and returns to its initial state. Once the user enters the PIN, the ATM requests the bank to verify the user's PIN. If the PIN is correct, the ATM displays the menu to the user within 5 seconds. If the PIN is incorrect, the ATM asks for correct PIN again. The user has to enter correct PIN in 3 attempts. If not, the ATM ejects the card and returns to its initial state. After the menu is displayed by the ATM, the user can choose from the available options e.g., deposit or withdrawl. For a withdrawl operation, the user has to enter the amount he wishes to withdraw. After entering the amount, the bank verifies the user details and returns a success message if the details are correct. Similarly,

the user enters the amount to deposit into account and then the ATM returns a success message if the deposit has been successful.

3.1 Formal Description of Mode Graphs and Timed Event Sequences

All the definitions and algorithms in this section are taken directly from [13].

3.1.1 Mode Graph

A *mode graph* is formally defined as a deterministic state machine in which the states are called modes and the transitions from one mode to the other are triggered by the events in the system. A *mode graph* is a tuple $\mathcal{M} = (M, m_0, m_f, \Sigma, T)$, where M is a finite set of modes, m_0 is the initial mode, m_f is the final mode (which can be identical to m_0), Σ is a set of events, and $T : M \times \Sigma \rightarrow M$ is a transition function. A triple (m_i, e_k, m_j) represents a transition between two modes m_i and m_j on an event e_k.

In the mode graph of Figure 3.1 there are fifteen different modes. m_0 *(card-not-inserted)* is both the initial mode and the final mode. An event *enter-pin* triggers the transition from *card-inserted* to *pin-entered*, and is respresented as *(card-inserted, enter-pin, pin-entered)*.

For any given two modes m_i and m_j, m_i is said to be the *dominating mode* of m_j iff all the paths to m_j from the initial mode in the mode graph pass through m_i [12]. We call this the *Dominance* relation and denote it as m_i *DOM* m_j. The same relation can be extended to events as well. We say that an event e is dominated by mode m_i iff, m_j is the source mode for e and $m_i DOM m_j$. For example, in the mode graph of Figure 3.1, the mode *card-inserted* is a dominating mode of *pin-entered* and event *request-data-for-bank* is dominated by all the modes which dominate the mode *user-verified*.

A clock variable t_i chosen from a set of clock variables $V = \{t_1, t_2....\}$, indicates the time of leaving mode m_i. If the mode m_i is part of a cycle then, t_i indicates the time of first event occurrence at mode m_i .

3.1.2 Timed Event Sequences

The scenarios describing the partial behaviours of a real-time system are expressed formally in the form of Timed Event Sequences (TES). A TES contains

1. The initial mode of the scenario,
2. The final mode of the scenario,
3. A set of events and their corresponding time annotations.

Each event in a TES represents an interaction between the system and the user or between the environment and the system or an internal action within the system. A TES is meaningless without a mode graph.

Given a mode graph $\mathcal{M} = (M, m_0, m_f, E, T)$, a Timed Event Sequence ξ is denoted by a tuple $\langle m^{initial}, \Psi, m^{final} \rangle$ where, $m^{initial}$ and m^{final} are the initial and final modes in the TES and $\Psi = [\psi_1, \psi_2, ..., \psi_n]$ is a non empty sequence of timed events of the form (e_i, σ_i), where $e_i \in E$ and σ_i is the set of time annotations corresponding to event e_i. A time annotation is of the form $W - t_j \sim a$, where W is the wall clock time, t_j is a time variable from V, $\sim \ \in \{\leqslant, \geqslant, <, >, =\}$ and $a \in \mathbb{Q}$. In $W - t_j \sim a$, t_j is the time of leaving a mode m_j, such that $m_j \ DOM \ m_i$. We call this assumption the *dominance assumption* and this assumption ensures the time variables are *well-defined*. That is, a time variable can only be used in a time annotation on a transition iff, that time variable is defined on every path to that transition.

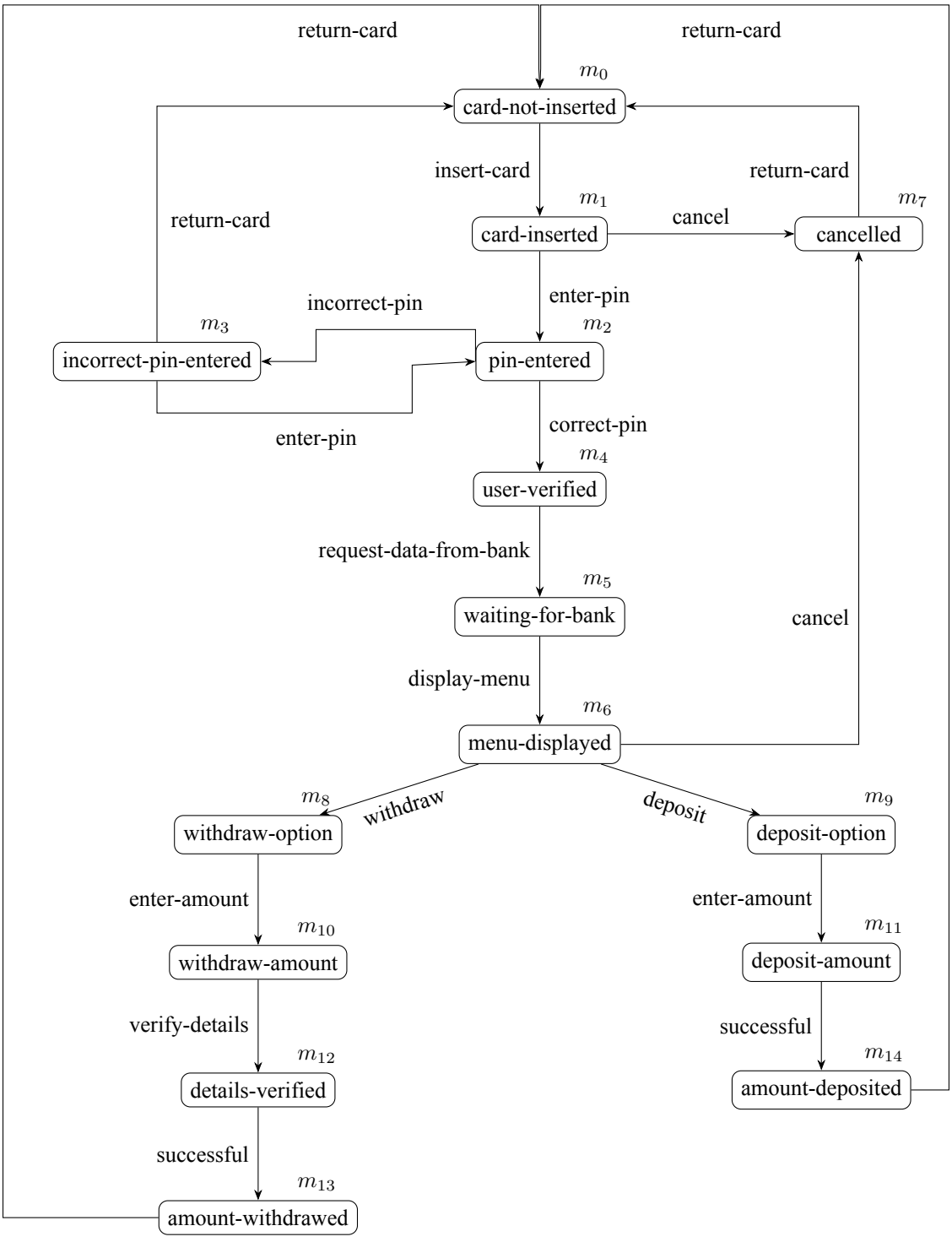


Figure 3.1: Mode Graph for ATM

For a TES ξ, the $initial\_mode(\xi) = m^{initial}$, $final\_mode = m^{final}$ and $events(\xi) = \Psi$. The set $modes(\xi)$ is the set of modes that can be reached from $initial\_mode(\xi)$ by performing some non-empty contiguous sequence of events in ξ.

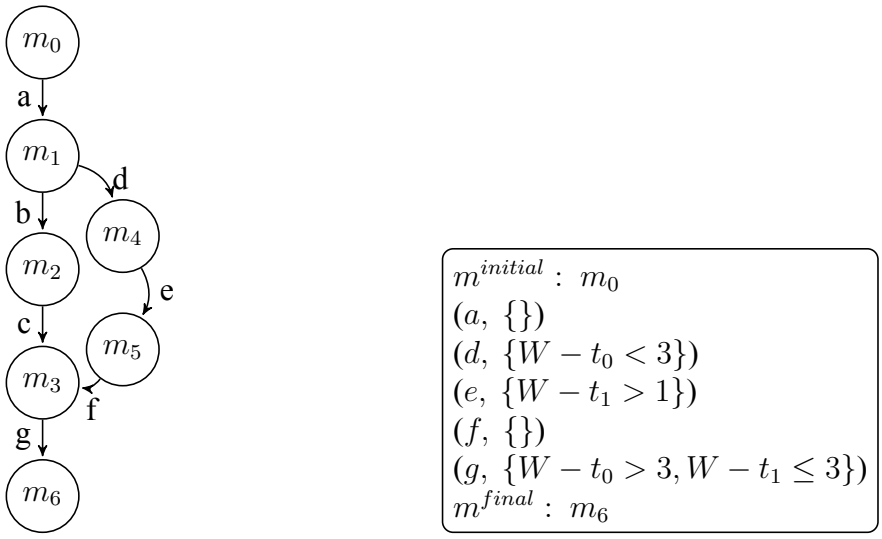


Figure 3.2: A mode graph and a scenario satisfying the dominance assumption

Figure 3.2 shows a mode graph and a scenario where the dominance assumption is satisfied. The initial mode of the mode graph is m_0 and the final mode is m_6. The modes m_0, m_1 and m_3 are dominating modes of m_6 and moreover, transition g is dominated by all the modes that dominate m_3. Notice that there are two paths to reach the final mode from the initial mode. The clocks t_0 and t_1, on transition g refer to the modes m_0 and m_1 respectively, which are the dominating modes of m_3.

The TES of scenario 1 in Figure 3.3 describes the behaviour of an ATM machine in which, the user enters an incorrect pin in the first attempt and later enters the correct pin. In the TES of scenario 2 the user enters the correct pin in his very first attempt. In scenario 1, the ATM waits in its initial mode $card - not - inserted$ until an user inserts the card. The user inserts the card at time t_0 and then, the system requests the pin. The user enters the pin at a time W such that the time difference between two events $insert - card$ and

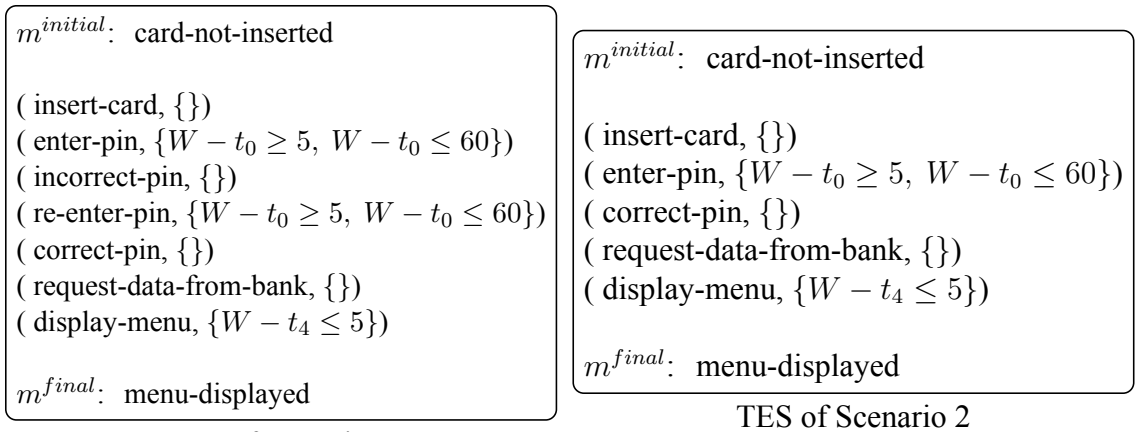


Figure 3.3: Two TES corresponding to the ATM machine

$enter - pin$ ($W - t_0$) is within $[5, 60]$ seconds. Upon receiving the pin, the system notifies that the pin is incorrect and then the user enters the correct pin. Then, the user is verified at a time t_4 and the menu is displayed to the user within 5 seconds, i.e., $W - t_4 <= 5$.

3.2 Generating Timed Automata from Scenarios

Our goal is to generate a deterministic timed automaton with minimal number of states from a given mode graph and a set of scenarios that are expressed as TES. The synthesis method includes two algorithms:

1. The first algorithm takes a set of TES and a mode graph as input and constructs a graph with states, transitions and time annotations.

2. The second algorithm takes the time annotated graph generated by the previous algorithm as input and assigns clocks, clock resets and clock constraints to transitions.

To construct a deterministic timed automaton with minimal number of states, the timed automaton should have *one initial state*, only one transition should be triggered at any given time and also, the timed automaton should be connected. A connected timed automaton

implies that, every state in the automaton is reachable from the initial state and that there should exist a path from every state to the final state.

We introduce four criteria that a set of TES $\{\xi_1, \xi_2, .., \xi_n\}$ must satisfy in order to make generation of such an automaton feasible. A set of scenarios expressed as TES that comply with these criteria are said to be *complete*.

Given a mode graph $\mathcal{M} = (M, m_0, m_f, \Sigma, T)$, a set of TES Ξ is *complete* if:

1. For every $\xi \in \Xi$, either $initial\_mode(\xi) = m_0$ or there exist $\xi_1, \xi_2, ..., \xi_j \in \Xi$, such that $\xi_j = \xi$ and:
   - $initial\_mode(\xi_1) = m_0$,
   - $initial\_mode(\xi_k) \in modes(\xi_{k-1})$, for each $\xi_k,\ 1 < k \leq j$.
2. For every $\xi \in \Xi$, either $final\_mode(\xi) = m_f$ or there exist $\xi_1, \xi_2, ..., \xi_j \in \Xi$, such that $\xi = \xi_1$ and :
   - $final\_mode(\xi_j) = m_f$,
   - for each $\xi_k,\ 1 < k \leq j$, $final\_mode(\xi_{k-1}) \in modes(\xi_k)$.
3. Every $\xi \in \Xi$ is *compatible with* $\mathcal{M}$: for each scenario ξ_i where $events(\xi_i) = e_1 e_2 \ldots e_n$, there is a partial run $s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_2} s_3 \ldots \xrightarrow{e_n} s_{n+1}$ of $\mathcal{M}$.
4. All TES in Ξ are *compatible with each other*: if two TES contain the same transition (between the same two modes), then the time annotations corresponding to the transition in both TES should not be mutually exclusive (which is easy to check, given the restricted form of constraints).

3.2.1 Constructing a Time Annotated Graph from Scenarios

Let P be a set of atomic propositions, strings, etc. A *time annotated graph* is a tuple of the form $G = (E, Q, q^0, q^f, R, L)$ where:

- E is a finite set of alphabets,
- Q is set of states,
- q^0 is the initial state,
- q^f is the final state,
- $R \subseteq Q \times Q \times E \times 2^{\phi(V)}$ is a set of transitions of the form (q, q', a, ϕ) where ϕ is the set of time annotations over V,
- $L : Q \rightarrow P_\perp$ is a function that maps each state to a label.

Given a mode graph $\mathcal{M}$ and a set of Timed Event Sequences $\{\xi_1, \xi_2, .., \xi_k\}$ as inputs, Algorithm 1 synthesizes a time annotated graph (TAG) G [13]. The algorithm initially starts with an empty TAG, G_0 and builds a partial graph G_1 using the first scenario ξ_1. The algorithm repeatedly takes a partially built graph G_k, and a scenario ξ_{k+1} $(1 < k < n)$ and then generates a new partial graph G_{k+1}. During the construction of G_{k+1} decision has to be made on whether to create new states and transitions. This is resolved with the help of state labels (modes). A new state s is created and labelled with a mode m_j if there is an event e from state q such that $L(q) = m_i$ and $(m_i, e, m_j) \in T$. New states and transitions are generated and added to the graph G_{k+1} if the transitions of ξ_{k+1} cannot be simulated.

The graph constructed by Algorithm 1 has the following properties:

1. It is acyclic: as we do not introduce a transition from a state to it's previous states.
2. Its graph is connected, because the input set of TES is complete.

Algorithm 1: Building states and transitions with time annotations from scenarios

Input : A mode graph $\mathcal{M} = (M, m_0, m_f, \Sigma, T)$, and a complete set of TES $\Xi = \{\xi_1, ..., \xi_n\}$

Output: Time-annotated graph $G_n = \langle E_n, Q_n, q^0, q^f, R_n, L_n \rangle$

$k := 0$; $E_k := \emptyset$; $Q_k := \emptyset$; $R_k := \emptyset$; $L_k := \emptyset$;

foreach *Timed Event Sequence* $\xi_i = \langle m_i^{initial}, \psi_1\psi_2...\psi_l, m_i^{final} \rangle$ *in* Ξ **do**

  $E_{k+1} := E_k$; $Q_{k+1} := Q_k$; $R_{k+1} := R_k$; $L_{k+1} := L_k$;

  // Find or create the first state for this scenario:

  if *there is a state such that* $L(s) = m_i^{initial}$ **then**

    $s_c :=$ the earliest such state; // "earliest": see property (3), p. 23

  else

    create a new state s;

    $Q_{k+1} := Q_{k+1} \cup \{s\}$;

    add label $m_i^{initial}$ to state s: $L_{k+1} := \{(s, m_i^{initial})\} \cup L_{k+1}$;

    $s_c := s$; // s_c always indicates the current source state.

  // Find or create the other states:

  foreach ψ_j *of the form* (e, ϕ), *where* $(m, e, m') \in T$ **do**

    if *there is a transition of the form* $r = (s_c, q, e, \phi')$ *in* R_{k+1} *and* $L(s_c) = m$ *and* $L(q) = m'$ **then**

      $\phi'' := \phi \wedge \phi'$; // see criterion 4 for completeness of scenarios

      $R_{k+1} := R_{k+1} \setminus \{r\} \cup \{(s_c, q, e, \phi'')\}$;

      $s_c := q$;

    else

      $E_{k+1} := E_{k+1} \cup \{e\}$;

      if *there is no state* $q \in Q_{k+1}$ *such that* $L(q) = m'$ *and* q *is not a predecessor of* s_c **then**

        create a new state q;

        $Q_{k+1} := Q_{k+1} \cup \{q\}$;

        add label m' to state q: $L_{k+1} := \{(q, m')\} \cup L_{k+1}$;

        create a new transition: $r := (s_c, q, e, \phi)$;

      else

        choose the earliest state q such that $L(q) = m'$ and q is not a predecessor of s_c; // "earliest": see property (3), p. 23

        create a new transition $r := (s_c, q, e, \phi)$;

      $R_{k+1} := \{r\} \cup R_{k+1}$;

      $s_c := q$;

  $k := k + 1$;

$q^f := f$, where f is the state with no outgoing transitions; // property (4)

3. By construction, two states have the same label only if one is a predecessor of the other. We create a new state with the same label only to avoid introducing a transition from a state to it's predecessor.

4. There must be at least one state with no outgoing transitions, because the graph is finite.

5. The graph is deterministic: we only add a new transition only if it does not exist from state s.

6. The graph is minimal, because:

   - We do not add additional states if the state with mode information m_i already exists. We only add in cases when the addition of a new transition creates a cycle.
   - In case of multiple existing states, we choose the state that was created first.

7. After construction, every scenario is a partial run of the constructed graph.

8. Every path in the final graph is identical to that of the mode graph because our algorithm adds a new state or transition based on the mode and transition information in the mode graph.

Given the two TES of Figure 3.3, the synthesized time annotated graph by Algorithm 1 is shown in Figure 3.4.

3.2.2 Constructing a Timed Automaton from Time Annotated Graph

After the time annotated graph is generated in the first step, we have to assign clock resets and clock constraints to the transitions to convert time annotated graph to a timed automaton.

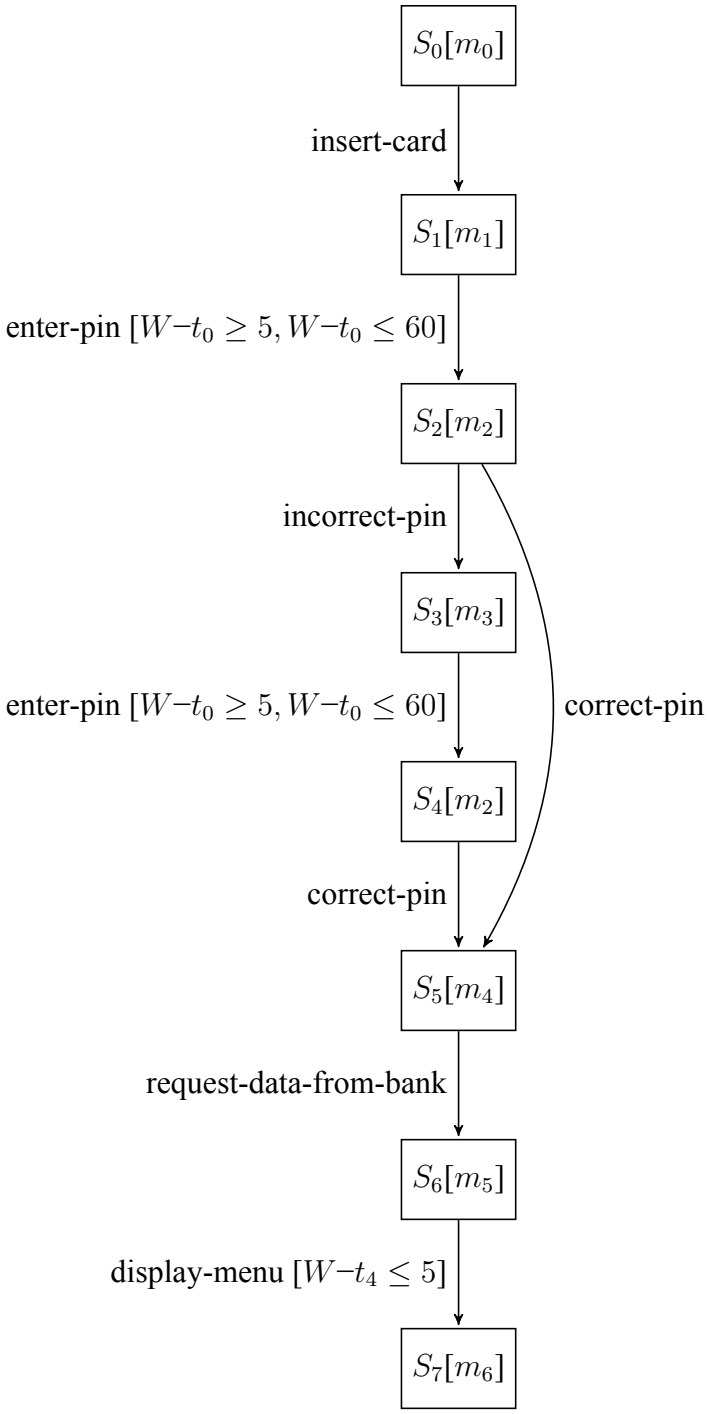


Figure 3.4: Time annotated graph synthesized from two TES in Figure 3.3

The three steps required to convert the constructed time annotated graph to a timed automaton are:

1. Determining the required number of clocks,

2. Adding clock resets,

3. Replacing the time annotations with the clock constraints.

Algorithm 2 performs the three steps mentioned above. During the execution of the algorithm, each transition of the graph is visited twice. At first, each transition is visited and a clock variable is mapped to each time variable that occurs in its time annotations. That is, a clock variable c_j is assigned to a time variable t_j on a transition, if the transition has a time annotation of the form $W - t_j \sim a$. At the end of first visit, we will have a list of all the clock variables that are mapped to their corresponding time variables. During the second visit of the graph, the algorithm will add clock constraints and add clock resets to the transitions wherever applicable. For example, if there is a time annotation $W - t_0 > 5$ on a transition r_1, then the clock constraint $c_0 > 5$ is added to r_1. This implies that, clock c_0 repalces the time variable t_0 in the time annotation. Because of our dominance assumption, whenever there is a time annotation $W - t_j \sim a$ that appears on a transition r, every path that leads to the transition r must have already passed through a state labelled m_j. So clock variable c_j mapped to t_j is initialized or reset before it is used on the transition r. Hence the clock c_j will be *well-defined* at r.

The timed automaton generated by applying Algorithm 2 to the time annotated graph of Figure 3.4 is shown in Figure 3.5.

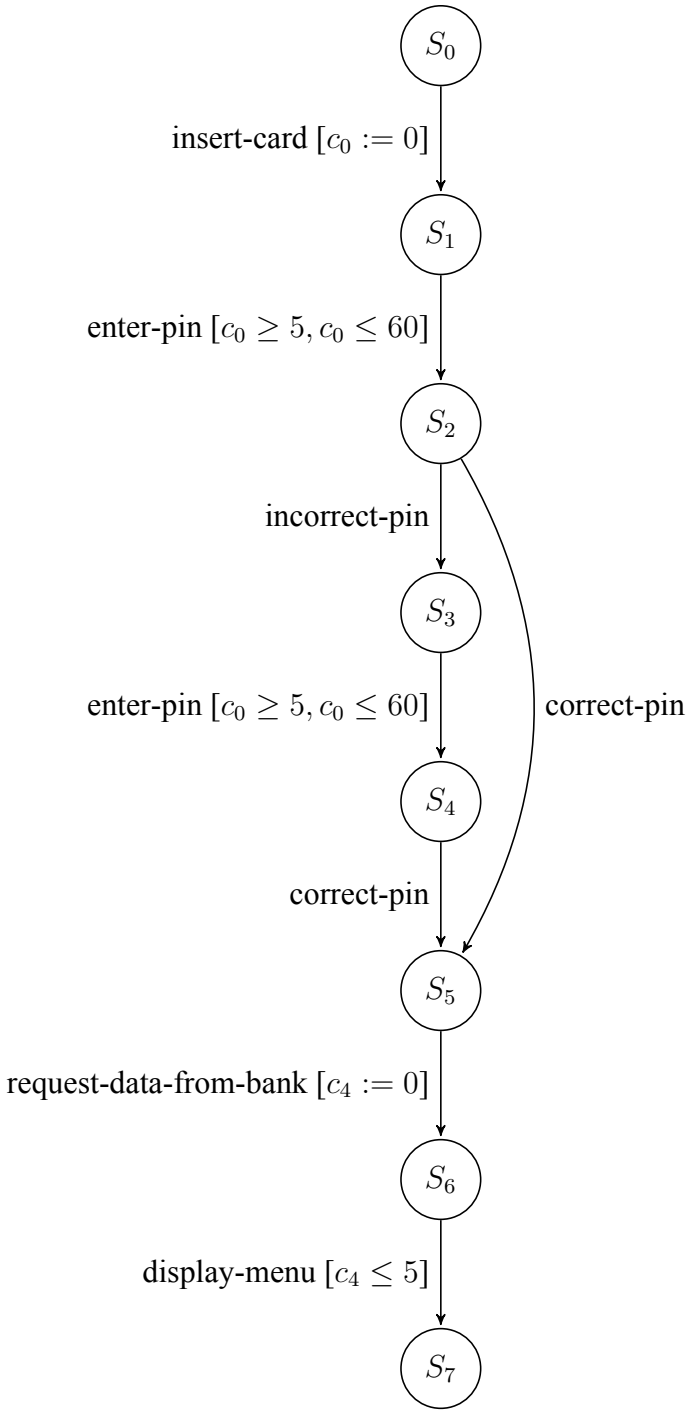


Figure 3.5: Timed automaton constructed from time annotated graph in Figure 3.4

Algorithm 2: Generating clock operations from a time annotated graph

Input : A time-annotated graph $G = \langle E, Q, q^0, q^f, R, L \rangle$
Output: A timed automaton $\mathcal{A} = \langle \mathcal{E}, \mathcal{Q}, \{\amalg'\}, \{\amalg^{\{}\}, \mathcal{C}, \times \rangle$

$C := \emptyset$; // the set of clocks
$\Theta := \emptyset$; // the set of transitions
$ca := \emptyset$; // the set of clock assignments
foreach *transition* $r = (s, q, e, \phi) \in R$ **do**
    foreach *time annotation* $W - t_i \sim a \in \phi$ **do**
        if $(t_i, c_i) \notin ca$ **then**
            generate a new clock c_i;
            $C := C \cup \{c_i\}$;
            $ca := ca \cup \{(t_i, c_i)\}$;
$\lambda := \emptyset$; // set of clocks to be reset in a transition
$\delta := \emptyset$; // set of clock constraints in a transition
foreach *transition* $r = (s, q, e, \phi) \in R$ **do**
    if $(t_i, c_i) \in ca$ *and* $L(s) = m_i$ **then**
        $\lambda := \{c_i\}$;
    foreach *time annotation* $W - t_j \sim a \in \phi$ **do**
        $\delta = \delta \cup \{c_j \sim a\}$, where $(t_j, c_j) \in ca$;
    $r_a = (s, q, e, \lambda, \delta)$;
    $\Theta := \Theta \cup \{r_a\}$;

4 Optimal Clock Allocation of Timed Automata

All the algorithms, definitions, lemmas and theorms along with their proofs appearing in this section are taken directly from [14].

In this section, we consider the problem of optimally allocating clocks to timed automata. The timed automaton constructed as a result of the synthesis method belongs to the class of timed automata that satisfies these properties:

1. The automaton has a unique initial state s_0. Every state is reachable from s_0,
2. A clock constraint on a transition 'r' can only refer to the times of transitions from states that dominate the transition 'r', we call this *dominance assumption*,
3. A clock t_j can only be reset on a transition leaving a state s, where label is j, that is $L(s) = j$.

The dominance assumption, guarantees that if a clock t_j is used in a constraint on some transition r, then t_j is well-defined on all the paths from the initial state to r. We also assume that, at most one clock can be reset on a transition. This assumption follows from the property (3) above. As clock t_j can only be reset on a transition leaving the state labelled j. The timed automaton of Figure 4.3 satisfies the properties mentioned above.

Reducing the number of clocks in a timed automaton is very crucial, as it directly affects the complexity of the verification problem. Given a timed automaton, it is undecidable to

check whether there exists another timed automaton that accepts the same language but has fewer clocks [9]. We propose an optimal clock allocation algorithm to the class of timed automata that satisfies the aforementioned properties. Our method does not change the graph of the original timed automaton. Moreover, the cost of our algorithm is quadratic in the size of the graph. The proposed algorithm reduces the number of clocks in a timed automaton whithout checking the satisfiability of clock constraints.

Our algorithm performs liveness analysis of clocks to determine on every transition which clocks are active. With the help of liveness analysis we can decide which clocks are no longer required and can be reused. A single new clock can be used to replace multiple clocks, only if the liveness ranges of those old clocks are disjoint. That is, the clock ranges do not share a transition. Consider the timed automaton of Figure 4.1. There are three clock resets $[t_0 := 0, t_2 := 0, t_4 := 0]$ on transitions r_0, r_2 and r_4 and three clock constraints $[t_0 < 2, t_2 < 2, t_4 < 2]$ on transitions r_1, r_3 and r_5 respectively. Clock t_0 is never used after transition r_1. So, t_0 can be re-allocated and re-assigned. Similarly t_2 is never used after t_4 is reset. In this automaton, all the three clocks can be replaced by a single clock e.g., c_0. The resulting automaton is shown in Figure 4.2.

Given a timed automaton $\mathcal{A}$, to transform it to an equivalent timed automaton $\mathcal{A}'$ with minimal number of clocks, we need to perform the following steps in order:

1. Calculate the liveness ranges of clocks in the timed automaton $\mathcal{A}$,
2. Replace the original clocks in $\mathcal{A}$ with a set of new clocks,
3. Rewrite the clock constraints and clock resets in $\mathcal{A}$ in terms of new clock variables.

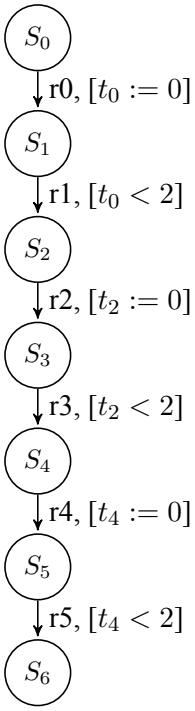


Figure 4.1: A simple timed automaton

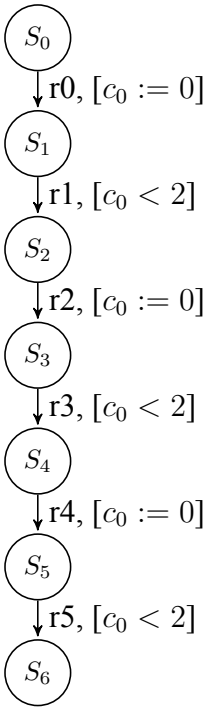


Figure 4.2: A simple optimally allocated timed automaton

4.1 Liveness Analysis of Clocks

Liveness analysis of clocks is performed by Algorithm 3. Before presenting the algorithm, we introduce some terms and definitions.

Let $\mathcal{A} = (E, Q, \{q^0\}, Q_f, V, R, L)$ be the timed automaton and $r = (s, s', e, \lambda_r, \phi_r) \in R$ be a transition. Let $N = \{j \mid t_j \sim a \in \phi \vee t_j \in \lambda$, where $(s, s', e, \lambda_r, \phi_r) \in R\}$, be a set of *clock numbers* used to denote subscripts of the clocks on all the transitions in R. Let $p = r_1...r_k$ be a path, we define $transitions(p) = \{r_1, ..., r_k\}$. The following are a set of functions used to calculate the liveness ranges.

- *clock_ref* : $R \rightarrow 2^N$ maps transition r to the set $\{j \mid t_j \sim a \in \phi_r\}$. Intuitively, *clock_ref*(r) is the set of clocks which are referred to in the clock constraints on r.
- *born* : $R \rightarrow 2^N$ maps transition r to the set $\{j \mid t_j \in \lambda_r$ and there exists a path $rr_1...r_k,\ k \geq 1$, such that $j \in$ *clock_ref*(r_k). Intuitively, *born*(r) identifies a clock that is reset on r whose value can be used on some transition reachable from r.
- *active* : $R \rightarrow 2^N$ maps transition r to the set $\{j \mid$ there is a path $rr_1...r_k$, $k \geq 1$, such that $j \in$ *clock_ref*(r_k)$\}$. Intuitively, *active*(r) identifies clocks that are "alive" on r (i.e., their values may be subsequently used). Notice that *born*(r) $\subseteq$ *active*(r).
- *needed* : $R \rightarrow 2^N$ maps transition r to *active*(r) $\cup$ *clock_ref*(r).

If there are any cycles in the graph then transitions r and r_k can be the same in the born and active definitions.

Both active and needed are important for liveness analysis as they help us to determine which clocks can be reused in the target timed automaton ($\mathcal{A}'$).

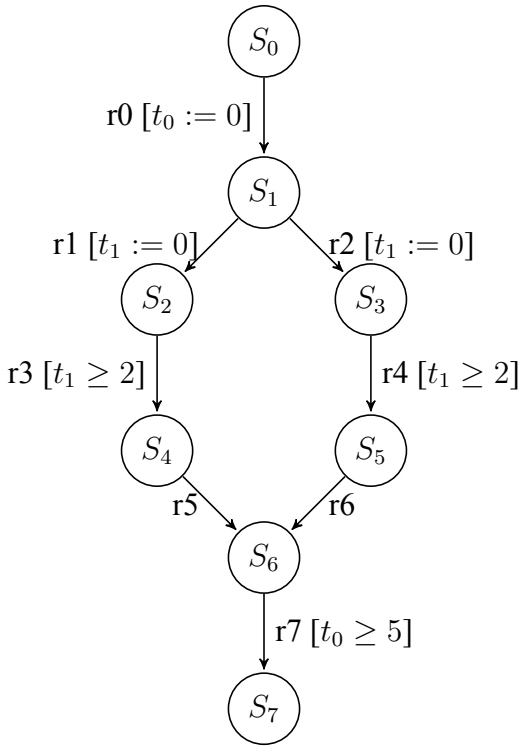


Figure 4.3: A timed automaton satisfying the dominance assumption

Definition 1. A path $p = r_0...r_n$ is a *path for clock* t_j iff $born(r_0) = \{j\}$ and $j \in needed(r_i)$ for $0 \leq i \leq n$.

In the automaton of Figure 4.3, there are two paths for clock t_0, $r_0r_1r_3r_5r_7$ and $r_0r_2r_4r_6r_7$ and two paths for clock t_1, r_1r_3 and r_2r_4.

Definition 2. $range : N \times R \rightarrow 2^R$ maps (j, r) to $\{r' \mid r' \in transitions(p)$, where p is a path for clock t_j that starts at $r\}$. Intuitively, *range* of t_j, where $j \in born(r)$, is the set of all transitions that belong to all the paths for clock t_j that begin at r.

In the automaton of Figure 4.3, clock t_1 will not be used in any path reachable from r_3 or r_4. So the range of t_1 ends at r_3 and r_4.

Given a timed automaton $\mathcal{A}$, Algorithm 3 determines the liveness ranges of the clocks in the automaton. It is a fixpoint iteration algorithm whose complexity is quadratic in the num-

ber of edges. It constructs an intermediate automaton which is an extension of $\mathcal{A}$ wherein, each transition is of the form $(r, born(r), active(r))$. For insance, in the automaton of Figure 4.3, transition r_1 will be extended to $(r_1, born = \{1\}, active = \{0, 1\})$. Similarly for transition r_7 the extended transition will be $(r_7, born = \emptyset, active = \emptyset)$.

Algorithm 3: Building the liveness ranges for clocks

Input : A timed automaton $\mathcal{A} = \langle E, Q, \{q^0\}, Q_f, V, R, L\rangle$.
Output: An extended timed automaton $\mathcal{A}_e = \langle E, Q, \{q^0\}, Q_f, V, R_e, L\rangle$, where R_e is the set of extended transitions.

$R_e := \emptyset$;
foreach *transition* $r = (s, q, e, \phi) \in R$ *in* $\mathcal{A}$ **do**
    $born(r) := active(r) := \emptyset$;
repeat
    foreach *transition* $r = (s, q, e, \lambda, \phi) \in R$ *in* $\mathcal{A}$ **do**
        foreach $r_o \in out(q)$ **do**
            $active(r) := active(r) \cup ((active(r_o) \cup clock\_ref(r_o)) \setminus born(r_o))$;
        if $L(s) = j$ *and* $j \in active(r)$ **then**
            $born(r) := \{j\}$;
        $R_e := R_e \cup \{(r, born(r), active(r))\}$;
until *there were no changes*;

The following lemma can be formulated from the above definitions.

Lemma 1. *For a timed automaton $\mathcal{A}$ with its set of states Q, we have*

$\forall_{q \in Q} \forall_{r_i, r_k \in in(q)}\ active(r_i) = active(r_k)$.

Proof. Assume $j \in active(r_i)$. Therefore there is a transition r in $out(q)$ such that $j \in needed(r)$ (i.e., t_j is referenced in r or one of its successors). But r can be reached from r_k, therefore $j \in active(r_k)$. The rest of the proof follows from symmetry. □

4.2 Clock Allocation

The liveness analysis algorithm generates the extended transitions of the original automaton. In the extended automaton, each transition has the $born$ and $active$ values associated with it. Here on, by transitions, we refer to the extended transitions. After finding the liveness ranges of clocks, we have to use those ranges to reuse the clocks and minimize the number of clocks in the final automaton $\mathcal{A}'$. Our method to minimize the number clocks based on liveness ranges of clocks revolves around the idea that: a clock can be reused if the active range of the clock has ended. The clock cannot be reused on a transition if the transition belongs to active range of that clock.

To understand the clock allocation algorithm and its properties, we introduce the following terminology:

Let A, B and C be sets and let $r \subset A \times B \times C$. The relation r can be applied to an argument $a \in A$ by using the operator ".": $r.a = \{(b, c) \mid (a, b, c) \in r\}$. Similarly, for $b \in B$, $r.a.b = \{c \mid (a, b, c) \in r\}$. The application operator associates to the left.

If, for every $(a, b) \in A \times B$, $r.a.b$ is either a singleton or the empty set, then r is a function of two arguments: $r : A \times B \to C$.

We assume P_0 to be a set of new clock variables, disjoint from V, and $|P_0| = |R|$. That is because, the number of clocks in a timed automaton $\mathcal{A}$ can at most be equal to the number of transitions in $\mathcal{A}$, since there is at most one reset per transition.

Definition 3. Given a timed automaton $\mathcal{A}$ with the set R of (extended) transitions and the set N of clock numbers, a *clock allocation* for $\mathcal{A}$ is a relation *alloc* $\subset R \times P_0 \times N$ such that $(r, c, j) \in$ *alloc* $\Rightarrow j \in$ *active*(r).

The presence of a tuple $(r, c, j) \in alloc$ implies that on transition r, the old clock t_j will be replaced with the new clock c in the final automaton.

Definition 4. A clock allocation *alloc* is *inconsistent* iff there exist two overlapping paths for some clock t_j, p and p' (which need not be different), some $c \in P_0$ and $r_k, r_l \in transitions(p) \cup transitions(p')$ such that $j \in \mathit{active}(r_k) \wedge j \in \mathit{active}(r_l) \wedge (r_k, c, j) \in \mathit{alloc} \wedge (r_l, c, j) \notin \mathit{alloc}$.
alloc is *consistent* iff it is not inconsistent.

Definition 5. A clock allocation *alloc* is *correct* if:

- *alloc* is a function, i.e., $\mathit{alloc} : R \times P_0 \to N$;
- *alloc* is consistent.

If the tuple $(r, c, j) \in \mathit{alloc}$, this means that, clock c is associated with clock t_j and clock t_j is active on transition r.

Definition 6. A clock allocation is *lean* if it is an injective function.

After allocating the clocks, a new clock c on transition r is not associated with more than one old clock.

Definition 7. The clock allocation *alloc* is *complete* iff, for every transition $r \in R$ and every $j \in \mathit{active}(r)$, there is a clock $c \in P_0$ such that $(r, c, j) \in \mathit{alloc}$.

Observation 1. Let $\mathcal{A}$ be a timed automaton with the set of transitions R, and let *alloc* be a complete, correct and lean clock allocation. Then the following holds:

$\forall_{r \in R} |\mathit{alloc}.r| = |\mathit{active}(r)|$.

Definition 8. We define the number of clocks used in an allocation by:

$\mathit{cost}(\mathit{alloc}) = |\{c \in P_0 \mid \exists_{r \in R} \exists_{j \in N} (r, c, j) \in \mathit{alloc}\}|$.

Definition 9. Let $\mathcal{A}$ be a timed automaton and let *alloc* be a complete correct clock allocation for $\mathcal{A}$. The allocation *alloc* is *optimal* if there is no complete correct allocation *alloc*$'$ for $\mathcal{A}$ such that $\mathit{cost}(\mathit{alloc}') < \mathit{cost}(\mathit{alloc})$.

Theorem 1. *Given a timed automaton $\mathcal{A}$ with the set of states Q and a correct complete clock allocation alloc, the following holds:*

$\forall_{s \in Q} \forall_{r_i, r_k \in in(s)}\ alloc.r_i = alloc.r_k$.

Proof. $alloc.r_i = alloc.r_k$ is equivalent to

$\forall_{j \in N} \forall_{c \in P_0}((r_i, c, j) \in alloc \Leftrightarrow (r_k, c, j) \in alloc)$.

Let j and c be such that $(r_i, c, j) \in alloc$. This implies that $j \in active(r_i)$. But then, from Lemma 1, we have $j \in active(r_k)$. Therefore there is some $r \in out(s)$ such that both $r_i r$ and $r_k r$ belong to paths for clock t_j (because j is needed at r). Since the paths overlap at r, from consistency of *alloc* we must have $(r_k, c, j) \in alloc$. The rest of the proof follows from symmetry. □

In the timed automaton of Figure 4.3, $alloc = \{(r_0, c_0, 0), (r_1, c_0, 0), (r_2, c_0, 0), (r_3, c_0, 0), (r_4, c_0, 0), (r_5, c_0, 0), (r_6, c_0, 0), (r_1, c_1, 1), (r_2, c_1, 1)\}$ is an example of a clock allocation for the automaton.

Before presenting our clock allocation algorithm, we explain a situation which needs a special care in order for the clock allocation in the final automaton to be consistent. In Figure 4.5, clock t_j is initialized on all transitions originating at s. But transitions r_1 and r_3 meet at state q and transitions r_2 and r_4 meet at state n. If clock t_j is assigned a different clock on each transition originating from s, then, the *alloc* values of all incoming transitions of states q and n will be different (Theorem 1 does not hold). So, the states q and n are *problematic states*. The clock allocation should be such that, same clock is assigned to j on all the outgoing transitions of state s that lead to the same problematic state q.

Definition 10. $Ranges : N \rightarrow 2^{2^R}$ maps j to $\{range(j, r) \mid j \in born(r)\}$. Intuitively, $Ranges(j)$ is the set of all the ranges for clock t_j.

Definition 11. Given a clock t_j, we define $rel_j = \{(a, b) \in Ranges(j) \times Ranges(j) \mid a \cap b \neq \emptyset\}$.

The relation rel_j is reflexive and symmetric; rel_j^* is its transitive closure.

Definition 12. Let $j \in N$ and $a \in \mathit{Ranges}(j)$. We define $\mathit{Rel}(j, a) = \{b \in \mathit{Ranges}(j) \mid a \; \mathit{rel}_j^* \; b\}$.

One can think of $\mathit{Rel}(j, a)$ as the set of ranges to which range a for clock t_j is directly or indirectly "related". The ranges are related, because they share some transitions, so on all of these ranges t_j must be represented by the same clock. Naturally, $b \in \mathit{Rel}(j, a)$ implies $\mathit{Rel}(j, a) = \mathit{Rel}(j, b)$.

Definition 13. Let $j \in N$ and $a \in \mathit{Ranges}(j)$. The set $\{r \in \bigcup_{S \in \mathit{Rel}(j,a)} S \mid j \in \mathit{active}(r)\}$ will be called a *family* for t_j.

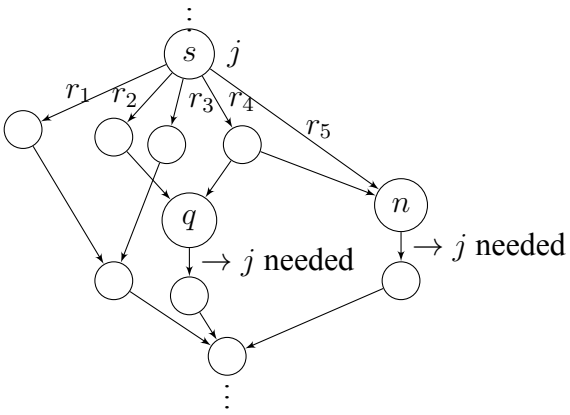


Figure 4.4: A timed automaton with problematic states and families

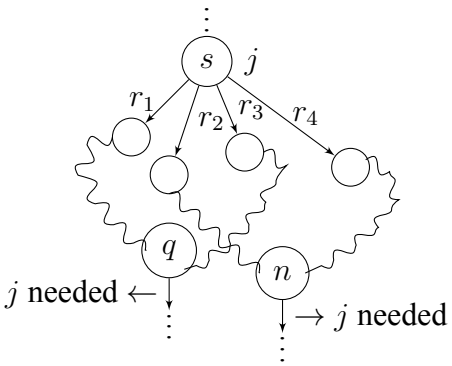


Figure 4.5: A timed automaton with problematic states

Let there be two transitions r_i, r_k from a state S such that, $L(S) = j$, i.e., $born(r_i) = j, born(r_k) = j$. These two transitions will belong to the same family F if there is a state S_p such that, S_p is reachable from r_i, r_k and clock j is needed in the outgoing transitions of S_p. Thus S_p is a problematic state. In Figure 4.4, transitions r_2 and r_4 belong to the same family and the state q is a problematic state. In figure 4.5, transitions r_1 and r_3 should belong to

one family and transitions r_2 and r_4 should belong to the same family. Thus, r_1, r_3 should be assigned the same clock. And r_2, r_4 should be assigned the same clock.

Observation 2. Let F be a family for some clock t_j, and let *alloc* be a complete correct allocation. Then there must exist a clock variable $c \in P_0$ such that $\mathit{alloc}.r.c = \{j\}$ for every transition $r \in F$ such that $j \in \mathit{active}(r)$. Otherwise *alloc* would be inconsistent. We say that c is *allocated* to F.

Observation 3. The number of families to which a transition r belongs is $|\mathit{active}(r)|$.

Proof. Assume the number of families to which transition r belongs is n. Therefore, there are exactly n families for different clocks that share transition r (two families for the same clock cannot share r, or they would be the same family). Since a family for some clock t_j has the property that j is active on all the transitions of the family, it follows that $\mathit{active}(r)$ contains exactly one element for each of the n families. □

Definition 14. Two families, F_1 and F_2, belong to the same *cluster* iff $F_1 \cap F_2 \neq \emptyset$. A cluster *cl* is a maximal set of such families, i.e., every family outside *cl* does not overlap with at least one of the families in *cl*.

We say transition r *belongs* to cluster $\mathcal{F}$, if there is a family $F \in \mathcal{F}$, such that $r \in F$.

Observation 4. Every family in a cluster must be allocated a different clock.

In the automaton of Figure 4.3, all the transitions belong to one cluster, and there are 2 families in the cluster. For clock $t_0 : \{r_0, r_1, r_2, r_3, r_4, r_5, r_6\}$ and for clock $t_1 : \{r_1, r_2\}$.

Definition 15. The *size* of a cluster is the cardinality of the set of families that form the cluster.

Theorem 2. *Let $\mathcal{A}$ be a timed automaton and alloc be a complete correct allocation for $\mathcal{A}$. Then $\mathit{cost}(\mathit{alloc})$ cannot be smaller than the size of the largest cluster in $\mathcal{A}$.*

Proof. This is a direct consequence of Observations 2 and 4. □

Before a detailed presentation of our clock allocation algorithm, we introduce two functions that are used in the algorithm.

- *reachable* : $Q \rightarrow 2^Q$ maps state q to the set of states that are reachable from q by some non-empty path.
- *reachable_from* : $Q \rightarrow 2^Q$ maps state q to the set of states from which it can be reached by some non-empty path.

In the automaton of Figure 4.3, *reachable*$(s_1) = \{s_2, s_3, s_4, s_5, s_6, s_7\}$ and *reachable_from* $(s_6) = \{s_0, s_1, s_2, s_3, s_4, s_5\}$.

All the new clocks are taken from the pool of new clock variables P_0.

The algorithm to optimally allocate a timed automaton is a sequence of procedures. We invoke the procedure $compute - allocation$ with the automaton obtained from Algorithm 3 and the set of available clocks in the pool P_0. Each state is tagged as $Unseen$ or $Seen$ or $Visited$. Initially all states are $Unseen$. When we visit the state for the first time, we try to annotate all it's immedeiate successors. Once all it's successors are annotated, the state becomes $Seen$. The state is tagged $Visited$ if all it's successors have been annotated and it is $Seen$. We start visiting the states from the initial state and then explore all the paths reachable from it. We will explore the complete automaton when we start from the initial state because of our assumption is that, every state is reachable from the initial state.

All the transitions emerging from a state ($out(s)$) are grouped into $mother$ and $other$ transitions. A transition $r \in mothers$ iff $\{r \in out(s) \mid j \in born(r)\}$. All the other transitions are grouped into $others$, i.e., $\{r \in out(s) \mid born(r) = \emptyset\}$. For example, in the automaton of Figure 4.3, r_0, r_1 and r_2 are mother transitions, whereas r_3, r_4, r_5, r_6 and r_7 are $other$ transitions. The partition of transitions into $mothers$ and $others$ helps us to handle problmatic states (if they exist).

If there is a problematic state s, then a clock t_j on all the transitions in $reachable\_from(s)$ should be assigned the same (new) clock c. A clock c is picked such that, it is the smallest clock available in the pool of clocks P_0. We partition all the transitions leading to a problematic state into sets of groups using the procedure $partition - into - set - of - groups$. All the transitions belonging to a group will be assigned the same clock c for a clock t_j. Initially, we add all the $mothers$ in $out(q)$[1] into a group. Then for each of the state s reachable from $target(r)$[2], we check if $L(q) \in active(in(s)$[3]$)$. If yes, then we add all such states into a set of problematic states PP. All the states in the set PP, which can be reached from more than two $mothers$ transitions are tagged as problematic states.

After the complete execution of the procedures *compute-allocation*, *annotate*, *visit*, *annotate-immediate-successors-of*, *propagate*, *partition-into-a-set-of-groups*, *find-clock*, each state of the target automaton has the following information:

- set of available clocks,
- set of clock assignments of the form (c, i). The tuple (c, i) implies that, a new clock c has been assigned to the old clock t_i in the target automaton.

During the execution of the algorithm, at each step, we annotate a state with all the available clocks in the pool and the clock assignments. We then carry forward those annotations to all the successors of the state. At each state, we check if any clock range has ended and then update the clock pool P_0 and assignements accordingly. A clock c on a transition r is freed and added back to the pool if there is an assignemt (c, j) such that, $(c, j) \in assigmnents(q)$ but $j \notin active(r_i)$ where $r_i \in out(q)$.

The runtime complexity of the algorithm is quadratic in the size of the graph. The only heavy operation is due to the presence of the routine $partition - into - set - of - groups$.

[1] $out(q)$ is the set of all transitions emanating from q.
[2] $target(r)$ is the state at which r ends.
[3] $in(s)$ is the set of all incoming transitions of s.

Wherein, for each transition $r \in mothers$, the routine looks upto $|Q|$ states to partition them. There are atmost $|R|$ tansitions and we lookup each transition in $mothers$ only once.

Procedure compute-allocation(timed automaton $\mathcal{A}_e$, set of clocks P_0)

Input : An extended timed automaton $\mathcal{A}_e = \langle E, Q, \{q^0\}, Q_f, V, R_e, L\rangle$ and the initial pool of available clocks, P_0.
Output: An extended timed automaton $\mathcal{A}'_e = \langle E, Q_e, \{q^0\}, Q_f, V, R_e, L\rangle$, where $Q_e = Q \times 2^{P_0} \times 2^{P_0 \times N}$.

```
foreach state s ∈ Q do
    Set the status of s to Unseen;
annotate(q^0, P_0, ∅);
visit(q^0);
```

Procedure annotate(state q, set of clocks p, set of assignments a)

```
// Invoked only when status of q is Unseen.
pool(q) := p;
assignments(q) := a;
Set the status of q to Seen;
```

Procedure visit(state q)

```
// Invoked only when the status of q is Seen or Visited.
if status of q is not Visited then
    Set the status of q to Visited;
    annotate-immediate-successors-of (q);
    foreach r ∈ out(q) do
        visit(target(r));
```

Procedure annotate-immediate-successors-of(state q)

```
Partition out(q) into mothers and others;
foreach r ∈ others do
    if status of target(r) is Unseen then
        propagate(q, r, ∅);
    // Otherwise target(r) is already properly annotated
if mothers ≠ ∅ then
    Groups := partition-into-a-set-of-groups(q, mothers);
    foreach group ∈ Groups do
        c := find-clock(q, group);
        foreach r ∈ group do
            // The target of r is Unseen (by the dominance assumption).
            propagate(q, r, {c});
```

Procedure propagate(state q, transition r, set of clocks sc)

```
// q is the source of r. Propagate pool(q) and assignments(q) to target(r), taking
  into account that some clock ranges
// may end on r. If sc is not empty, it must be a singleton: in that case assign its
  member to clock number L(q).
// Invoked only when the target of r is Unseen.
freed_assignments := {(d, j) | (d, j) ∈ assignments(q) ∧ j ∉ active(r)};
freed_clocks := {d | (d, j) ∈ freed_assignments};
tmp_pool := pool(q) ∪ freed_clocks;
tmp_assignments := assignments(q) \ freed_assignments;
if sc ≠ ∅ then
    tmp_pool := tmp_pool \ sc;
    tmp_assignments := tmp_assignments ∪ {(c, L(q))}, where c ∈ sc;
annotate(target(r), tmp_pool, tmp_assignments);
```

The following theorems hold for the algorithm:

Theorem 3. *The computed allocation is correct and lean.*

Proof. Observe that all the paths for a clock t_j begin at the same state. The initial transitions of these paths, i.e., the "mother" transitions, are prtitioned into groups. The "mother" transitions belonging to a group are exactly the initial transitions of a family for t_j. The algorithm associates some clock with a group: the association is propagated to all the tran-

Procedure partition-into-a-set-of-groups(state q, set of transitions *mothers*)

```
mother_targets := {target(r) | r ∈ mothers};
// Initially, each mother is in its own group.
Groups := ∅;
foreach r ∈ mothers do
    Groups := Groups ∪ {r};
PP := ∅;      // potentially problematic states
foreach r ∈ mothers do
    foreach s ∈ reachable(target(r)) do
        if L(q) ∈ active(r'), where r' is an arbitrary transition of in(s) then
            PP := PP ∪ {s};
// Those states in PP that can be reached from more than one mother are the
  problematic states.
foreach s ∈ PP do
    targets := reachable_from(s) ∩ mother_targets;
    Merge those members of Groups that contain transitions whose target is in
      targets;
return Groups;
```

Procedure find-clock(state q, set of transitions *group*)

```
// Find a clock for L(q) on transitions in group.
live_on_entry := {j | (c, j) ∈ assignments(q)};
dying_all := ⋂_{r∈group} (live_on_entry \ active(r));
// The set of clocks whose liveness ranges end in all the transitions in group:
released_all := {c | (c, j) ∈ assignments(q) ∧ j ∈ dying_all};
available := released_all ∪ pool(q);
Return the clock variable with the smallest number in available;
```

sitions of the paths for t_j that begin in the group, therefore there is no pair of transitions that satisfies the definition of inconsistency.

When some clock c is assigned to j on the transitions of a group, t_j is the only clock that is born, so c is not, on those transitions, assigned to any i such that $i \neq j$. Moreover, after c is assigned to j, it is removed from the pool and returned only on transitions on which t_j is not active. Therefore c cannot be assigned to any other i on any transition r such that $j \in \mathit{active}(r)$. So the allocation is always an injective function, i.e., it is lean. □

Lemma 2. *Assume alloc is a complete, correct and lean allocation. Then, for any transition r, the number of clocks in alloc.r is not greater than the size of the largest cluster to which r belongs.*

Proof. Assume cluster $\mathcal{F}$ with size n is the largest cluster to which r belongs. Assume that $|alloc.r| > n$. By Observation 1, $|active(r)| = n' > n$. By Observation 3, the number of families to which r belongs is n'. This implies that there is a cluster $\mathcal{F}'$, whose size is n', to which r belongs. But this contradicts the assumption that $\mathcal{F}$ is the largest cluster that includes r. □

Theorem 4. *The computed allocation is optimal.*

Proof. This is a direct consequence of Lemma 2, Theorem 2, Theorem 3 and the fact that the algorithm always allocates that of the available clocks which has the smallest number, i.e., a new clock is added to the set of used clocks only when none of those already in the set will do. □

5 Case Studies

Our results include, synthesizing a timed automaton from a given set of scenarios and optimally allocating clocks in the resulting timed automaton. Our synthesis alogirthm takes user scenarios expressed in Timed Event Sequences and a mode graph as input and generates a minimal, deterministic and acyclic timed automaton. The optimal clock allocation algorithm takes the synthesized automaton and optimally allocates the clocks.

To illustrate our approach for synthesizing timed automata and optimally allocating clocks to them, we present several real-world examples in the next few sub sections.

5.0.1 Automated Teller Machine

We use an invariant of an Automated Teller Machine (ATM) to illustrate our synthesis algorithm. The mode graph of the ATM is shown in Figure 5.3. In its initial state, the ATM machine waits for a user to insert his bank card. Once the card is inserted, the user can either cancel the transaction within 4 seconds or enter his PIN within 5 to 60 seconds. If the transaction is cancelled, the ATM ejects the card within 2 to 3 seconds and returns to it's initial state. Once the user enters the PIN, the ATM requests the bank to verify the user's PIN. If the PIN is correct, the ATM displays the menu to the user within 5 seconds. If the PIN is incorrect, the ATM asks for correct PIN again. The user has to enter correct PIN in 3 attempts. If not, the ATM ejects the card and returns to its initial state. The two TES shown in Figure 5.1 describe the partial behaviour of the ATM system explained above. The timed automaton synthesized from these two scenarios is shown in Figure 5.2.

$m^{initial}$: card-not-inserted

(insert-card, {})
(enter-pin, $\{W - t_0 \geq 5,\ W - t_0 \leq 60\}$)
(incorrect-pin, {})
(re-enter-pin, $\{W - t_0 \geq 5,\ W - t_0 \leq 60\}$)
(correct-pin, {})
(request-data-from-bank, {})
(display-menu, $\{W - t_4 \leq 5\}$)

m^{final}: menu-displayed

TES of Scenario 1

$m^{initial}$: card-not-inserted

(insert-card, {})
(enter-pin, $\{W - t_0 \geq 5,\ W - t_0 \leq 60\}$)
(correct-pin, {})
(request-data-from-bank, {})
(display-menu, $\{W - t_4 \leq 5\}$)

m^{final}: menu-displayed

TES of Scenario 2

Figure 5.1: Timed Event Sequences of the ATM

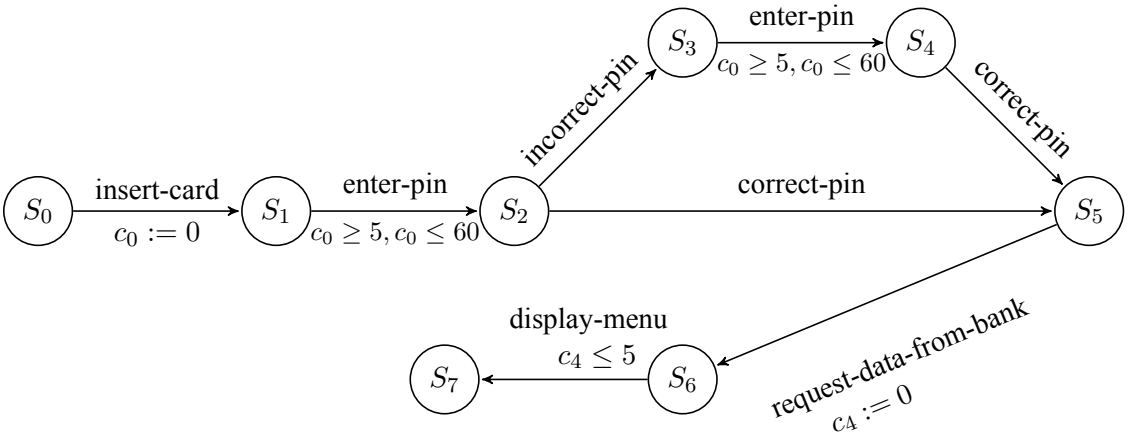


Figure 5.2: The timed automaton synthesized from Scenario 1 and Scenario 2

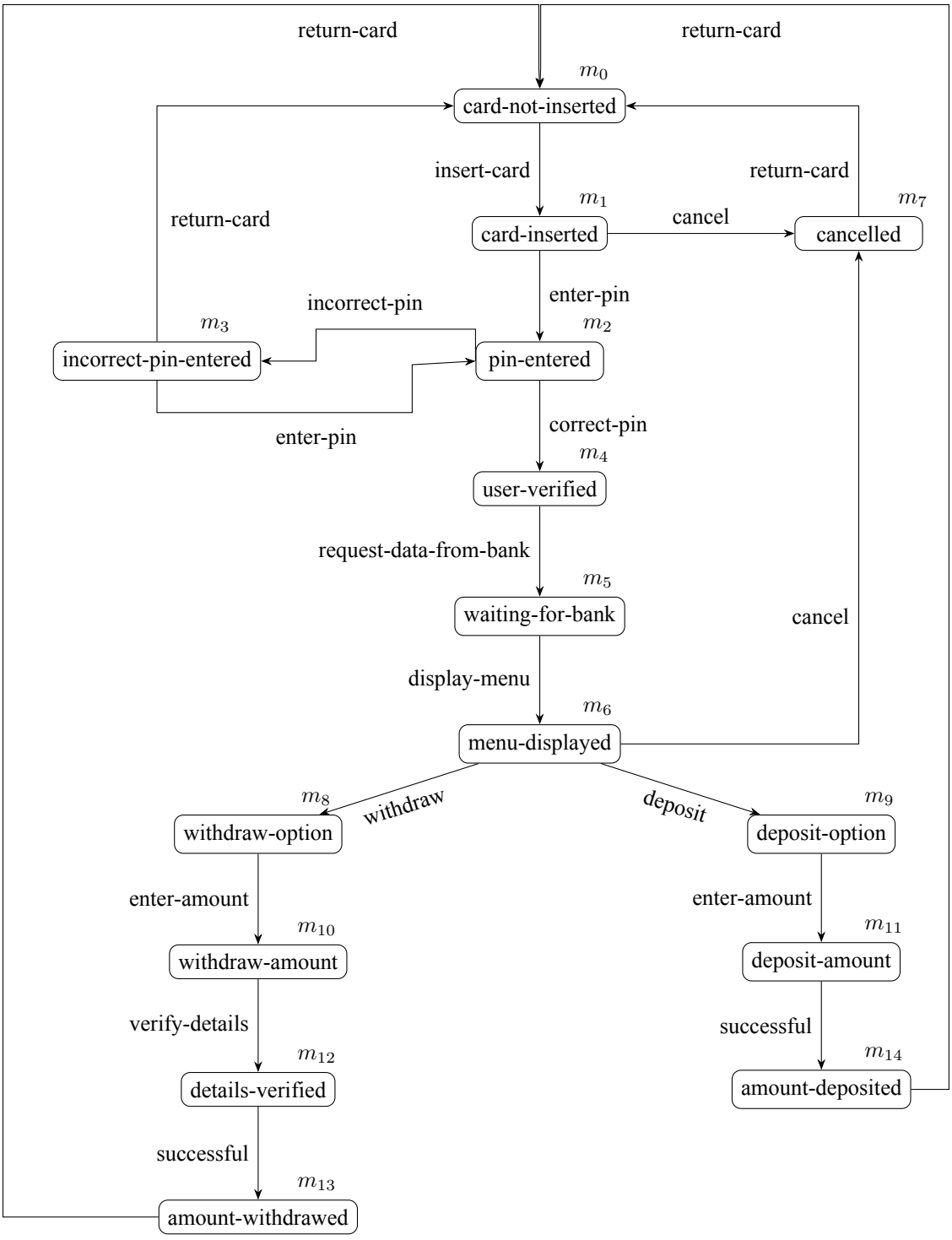
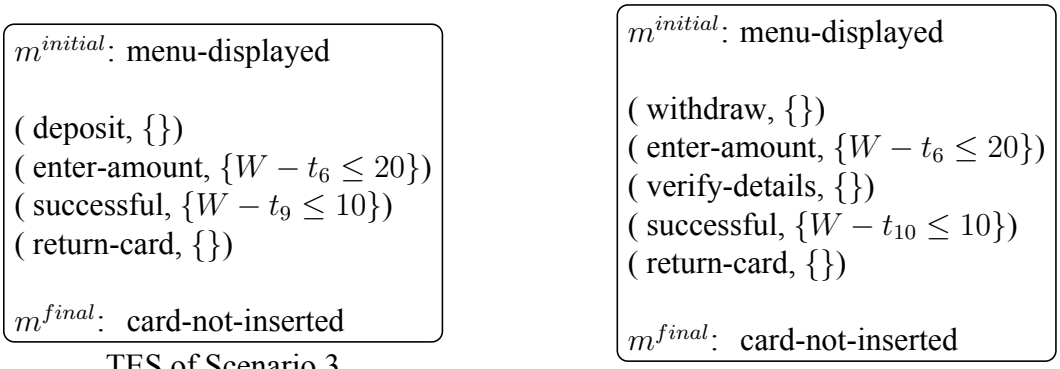


Figure 5.3: Mode graph of the ATM

Now consider an extended behaviour of the ATM machine. After the menu is displayed by the ATM, assume that the user can choose from the two options available: $deposit$ or $withdraw$. For a withdrawl operation, the user has to enter the amount he wishes to withdraw. After entering the amount, the bank verifies the user details and returns a success message if the details are correct. Similarly, the user enters the amount to deposit into account and then the ATM returns a success message if the deposit has been successful. The TES representing the $withdraw$ and $deposit$ operations are shown in Figure 5.4. The timed automaton synthesized by using the mode graph in Figure 5.3 and the TES in Figures 5.1 and 5.4 is shown in Figure 5.5.



TES of Scenario 3

TES of Scenario 4

Figure 5.4: Timed Event Sequences of the ATM with withdraw and deposit options

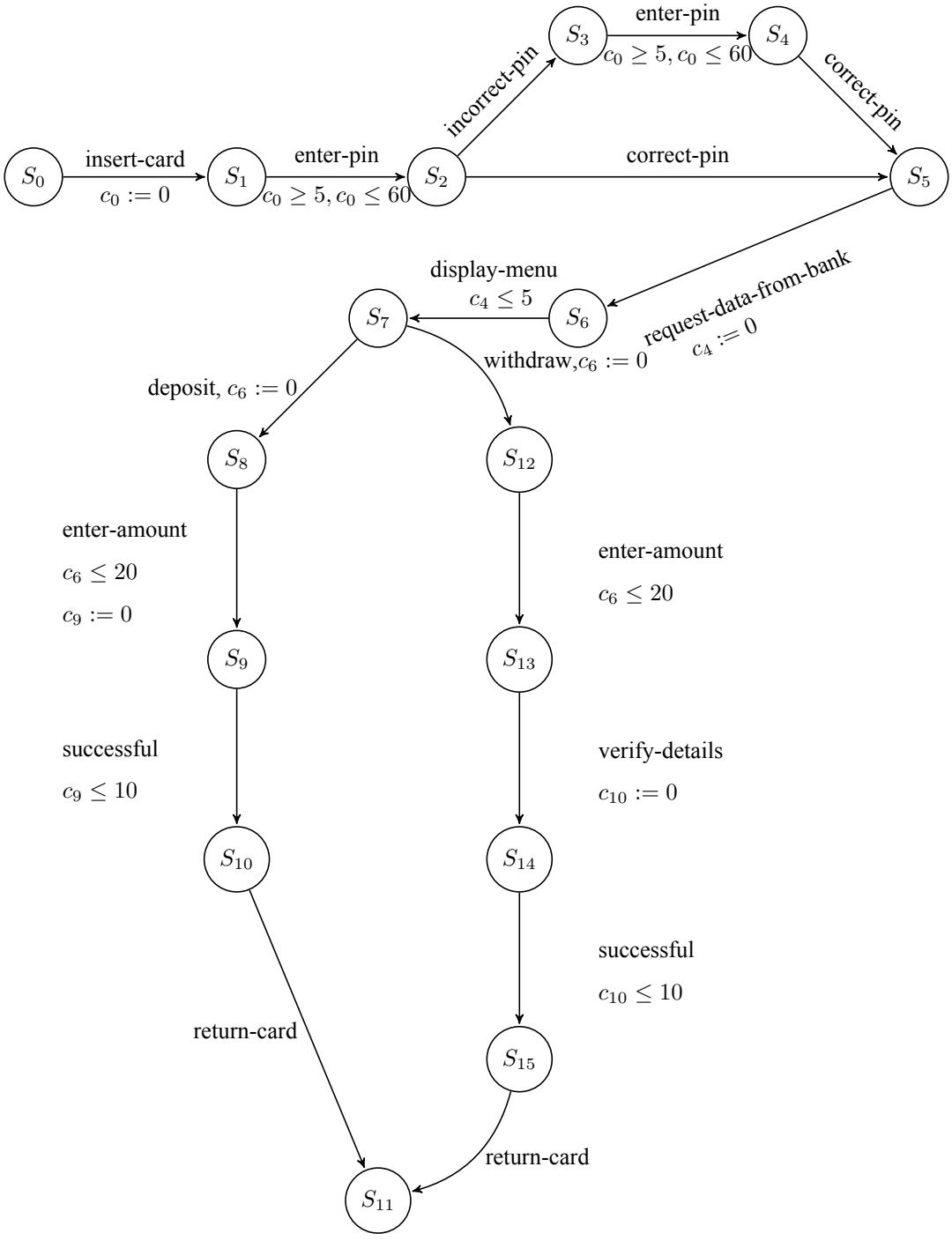


Figure 5.5: The synthesized timed automaton of the ATM

5.0.2 Light Control System

Conisder the variant of a light control system [3] shown in figure 5.6. Initially, the system is in $Idle$ state. The state changes from $Idle$ to $Light$ upon issuing the command ON. If ON is issued again within 3 units of time, the light will go to $Bright$ state. Else the light goes back to $Idle$ state. In the mode graph, $Idle$ is both the initial and final mode. The partial behaviour of the control system using which the model is generated is also given in Figure 5.7. The timed automaton synthesized by our algorithm using the mode graph in Figure 5.6 and the three TES in Figure 5.7 is shown in Figure 5.8.

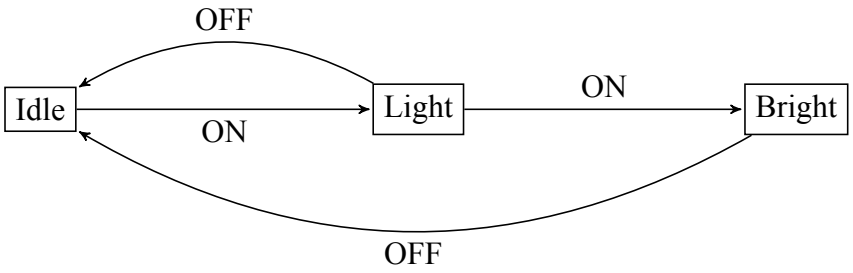


Figure 5.6: Mode graph of the Light Control System

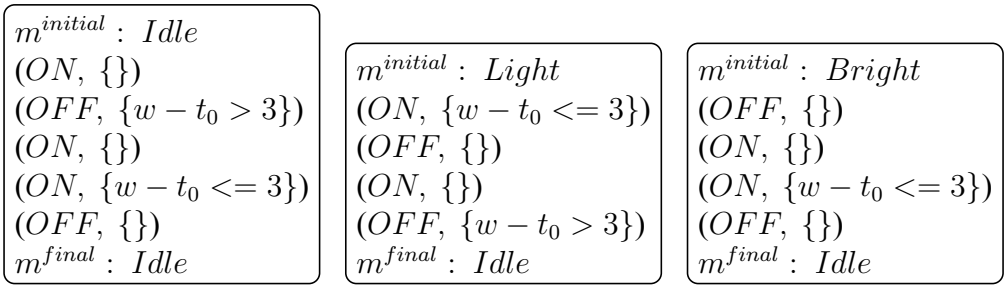


Figure 5.7: Timed Event Sequences of the Light Control System

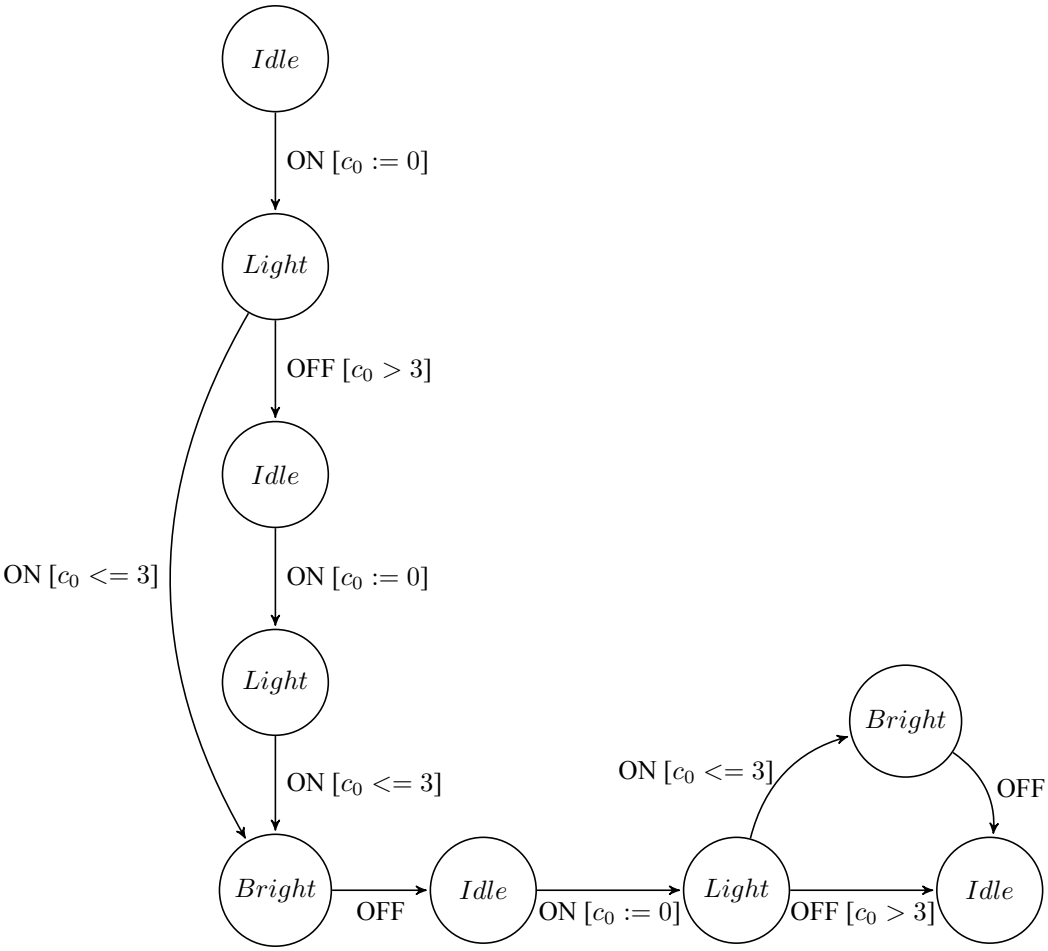


Figure 5.8: Timed automaton of the Light Control System

5.0.3 Fuel Management System

Consider the variant of a fuel management system [7] with three states - $Empty$, $Normal$, $Full$. Upon the action IN, the fuel is filled in and upon action OUT the fuel is used up. The mode graph of Figure 5.9 depicts one such system. In the mode graph, the initial mode is $Empty$ and the final mode is $Normal$. The timed automaton synthesized by our algorithm using the mode graph in Figure 5.9 and the three TES in Figure 5.10 is shown in Figure 5.11.

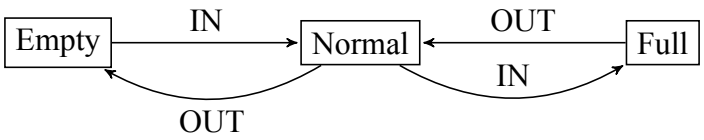


Figure 5.9: Mode graph of the Fuel Management System

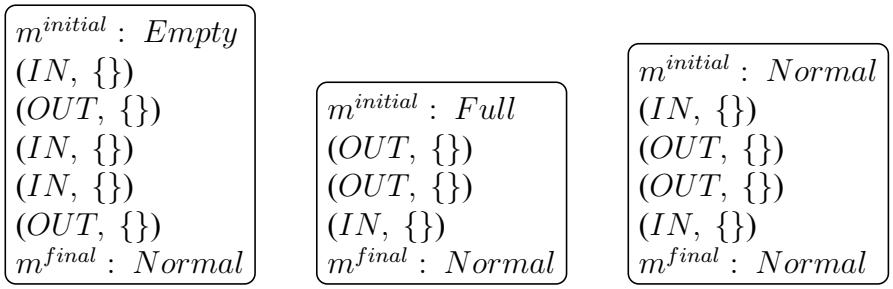


Figure 5.10: Timed Event Sequences of the Fuel Management System

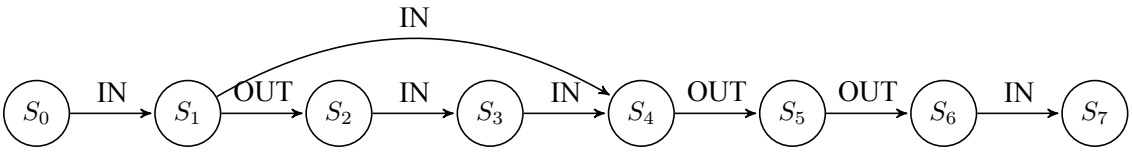


Figure 5.11: Timed automaton of the Fuel Management System

5.0.4 Train and Railroad Crossing System

We consider a variant of the Train component of the Train, Gate and Controller system [2]. Initially, the train is Far from the railroad crossing and the events $Enter$ and $Exit$ represent the events of entry and exit of the railroad crossing. The mode graph of such system is shown in Figure 5.12. The initial mode and the final mode in the mode graph is Far. The synthesized automaton using three TES in Figure 5.13 is shown in Figure 5.14.

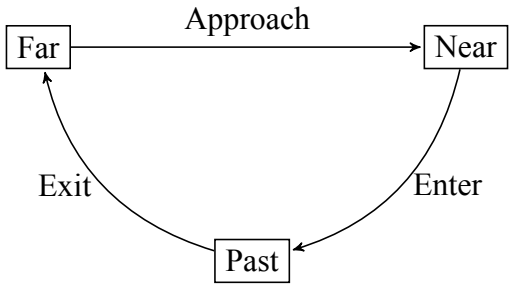


Figure 5.12: Mode graph of the Train And Railroad Crossing System

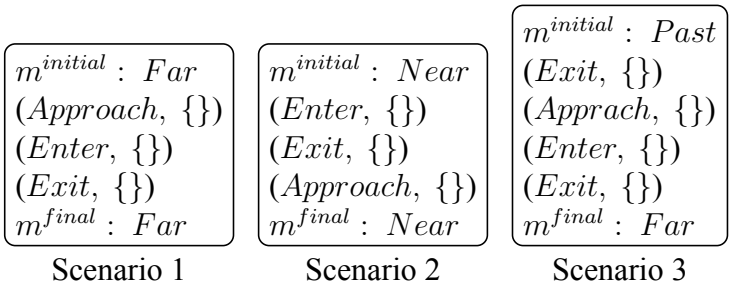


Figure 5.13: Timed Event Sequences of the Train And Railroad Crossing System

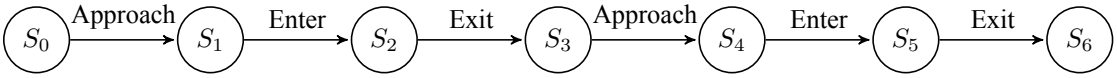


Figure 5.14: Timed automaton of the Train And Railroad Crossing System

In the next few subsections, we present several examples of how our optimal clock allocation algorithm is applied to real world examples.

5.0.5 Traffic Light System

Consider the example of the behaviour of a traffic light [8]. Figure 5.15 shows a variant of the traffic light model. The system periodically moves from $Initial$ state to Red, from Red to $Yellow$, from $yellow$ to $Green$ and then it is $Reset$. As seen from the automaton, on the transitions of $turn - yellow$ and $turn - green$ a clock constraint is checked and a new clock is born. Clearly, clocks are not optimally allocated in the timed automaton, as at any point of time only one clock will be sufficient. The result of applying our optimal clock allocation algorithm to this timed automaton is shown in figure 5.16. One clock is used instead of three clocks. That is, clock c_0 is assigned to t_0, t_1 and t_2. The clock assignments are as follows: $\{(c_0, t_0), (c_0, t_2), (c_0, t_1)\}$.

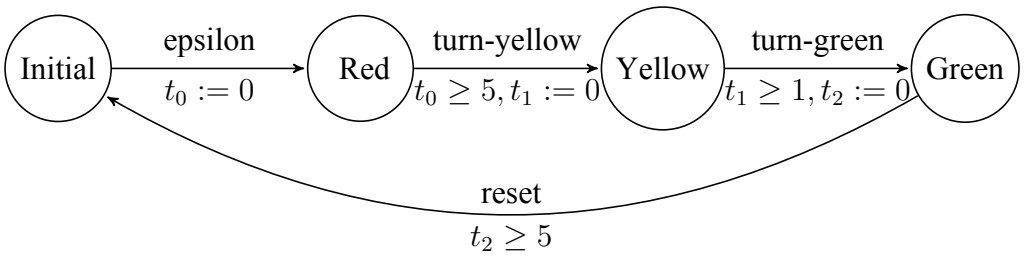


Figure 5.15: Timed automaton of the Traffic Light

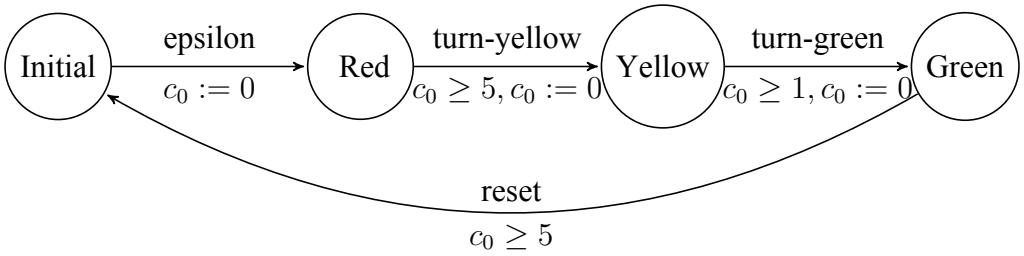


Figure 5.16: The optimally allocated timed automaton of the Traffic Light

5.0.6 CSMA/CD Protocol

Consider a variant of the CSMA/CD protocol [18] shown in Figure 5.17. Assume there are n senders $S_1, S_2, .., S_n$ and a medium M. The action $send$ means messages are sent, $busy$ means the channel is busy and cd indicates a colloision. $begin$ and end signifiy the beginning and the ending of a transmission. In the automaton, there are three clocks but at any point only two clocks are required. The range of clock x_1 ends on transition r_7. The clock can be reassigned to x_3 after the constraint is checked. The final optimally allocated timed automaton is shown in Figure 5.18

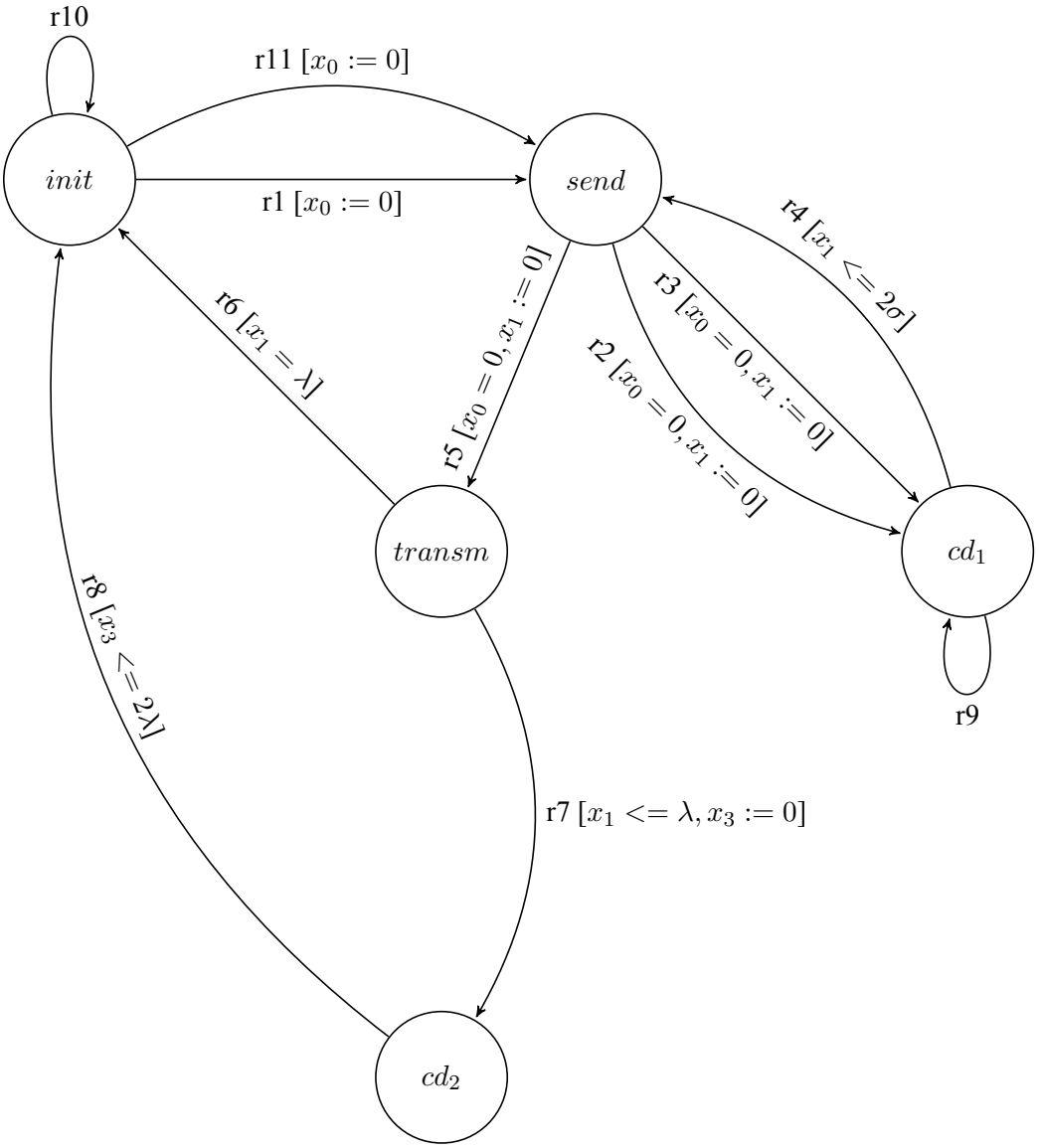


Figure 5.17: The timed automaton for the sender in CSMA/CD protocol

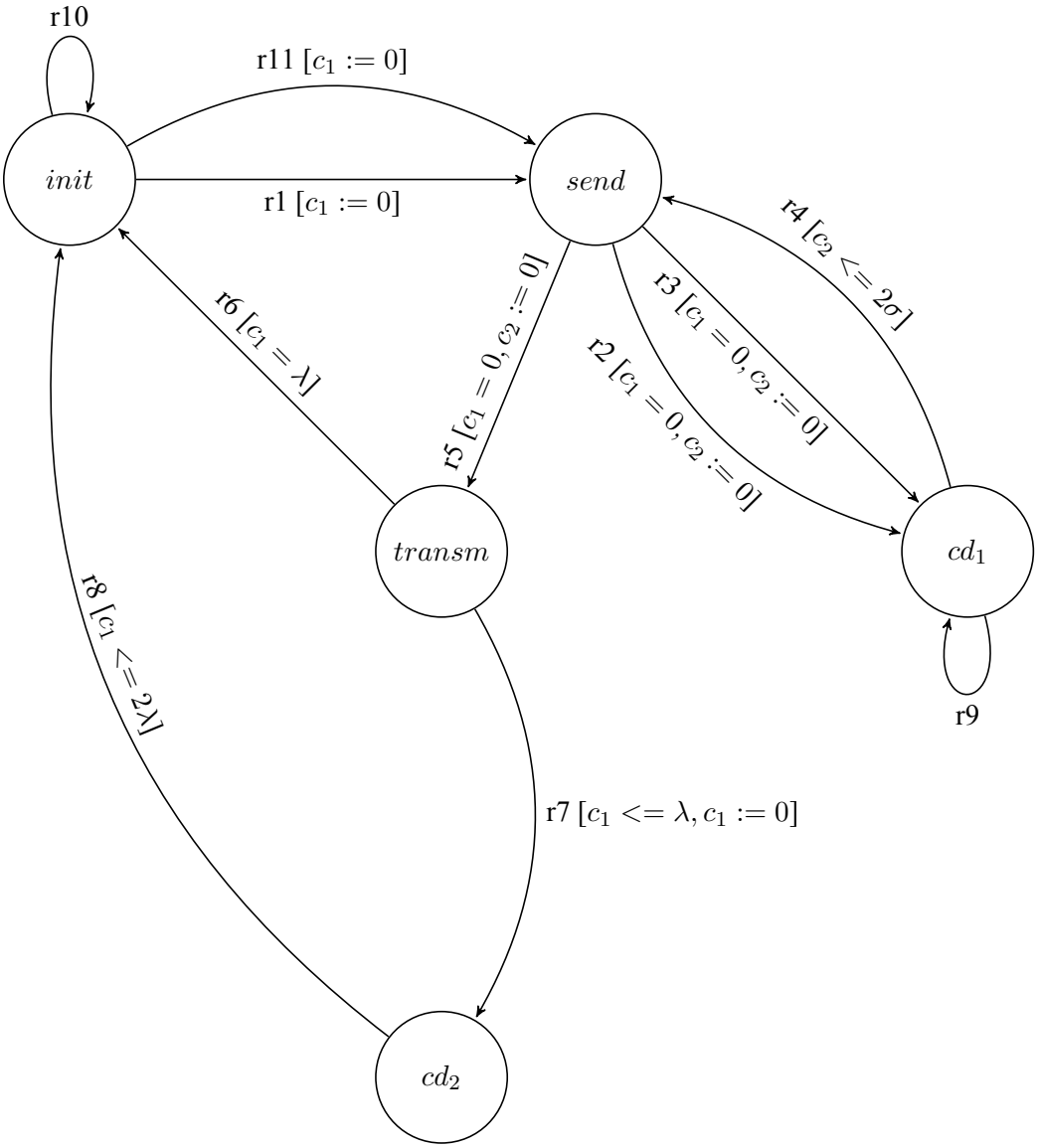


Figure 5.18: The optimally allocated timed automaton for the sender in CSMA/CD protocol

6 Conclusions

We develop and implement an algorithm to construct a minimal, deterministic and acyclic timed automaton from a given set of scenarios and a mode graph. A scenario is a partial behavior of the system during some time interval. We propose Timed Event Sequences to formally express the scenarios. The use of mode graphs helps us to understand what events are allowed in the automaton. We first synthesize a time annotated graph from Timed Event Sequences and a mode graph. We start with an empty graph and then extend the graph as we come across each scenario. The generated time annotated graph is then used as an input to the algorithm which assigns clocks, clock resets and clock constraints to the transitions. The generated timed automaton belongs to a class of timed automata that satisfies the dominance assumption. In this class, we also assume that, a clock t_i can be reset only on a transition leaving a state s such that $L(s) = i$.

For optimal clock allocation, our algorithm takes in a timed automaton $\mathcal{A}$ constructed using our synthesis method and generates an optimally allocated timed automaton $\mathcal{A}'$ where the number of clocks is minimal. We only consider the clock variables in clock constraints to minimize the number of clocks but not the satisfiablity of the clock constraints. To minimize the number of clocks, we perform liveness analysis on the clocks to determine which clocks can be reused. After performing liveness analysis, each transition is extended with $born$ and $active$ sets. The automaton with the extended transitions is used to minimize and optimally allocate the clocks. We do not change the graph of the original timed automaton and the complexity of the algorithm is quadratic in the size of the graph.

In the future, we will identify a more general class of timed automata to which our clock allocation method can be applied and extend our results to timed automata in general, not just the one that satisfies our dominance assumption.

Bibliography

[1] E. Abraham. *Modeling and Analysis of Hybrid Systems.* Faculty of Mathematics, Computer Science, and Natural Sciences RWTH Aachen University, 2012 (cit. on p. 12).

[2] R. Alur and D. L. Dill. "A Theory of Timed Automata". In: *Theor. Comput. Sci.* 126.2 (Apr. 1994), pp. 183–235. ISSN: 0304-3975. DOI: 10.1016/0304-3975(94)90010-8. URL: http://dx.doi.org/10.1016/0304-3975(94)90010-8 (cit. on pp. 1, 4, 5, 7, 53).

[3] P. Bouyer. *An introduction to timed automata*. 2011-2012. URL: https://pdfs.semanticscholar.org/a363/3e789b4e17f4eafe1868605911bea49d2be0.pdf (cit. on p. 50).

[4] E. M. Clarke Jr., O. Grumberg, and D. A. Peled. *Model Checking*. Cambridge, MA, USA: MIT Press, 1999. ISBN: 0-262-03270-8 (cit. on pp. 9–11).

[5] C. Damas, B. Lambeau, F. Roucoux, and A. van Lamsweerde. "Analyzing Critical Process Models Through Behavior Model Synthesis". In: *Proceedings of the 31st International Conference on Software Engineering*. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 441–451. ISBN: 978-1-4244-3453-4. DOI: 10.1109/ICSE.2009.5070543. URL: http://dx.doi.org/10.1109/ICSE.2009.5070543 (cit. on p. 2).

[6] C. Daws and S. Yovine. “Reducing the number of clock variables of timed automata”. In: *17th IEEE Real-Time Systems Symposium*. Dec. 1996, pp. 73–81. DOI: 10.1109/REAL.1996.563702 (cit. on p. iii).

[7] P. Derler, E. A. Lee, and A. S. Vincentelli. “Modeling Cyber-Physical Systems”. In: *Proceedings of the IEEE* 100.1 (Jan. 2012), pp. 13–28. ISSN: 0018-9219. DOI: 10.1109/JPROC.2011.2160929 (cit. on p. 52).

[8] A. Dubey, S. Nordstrom, T. Keskinpala, S. Neema, T. Bapty, and G. Karsai. “Towards a verifiable real-time, autonomic, fault mitigation framework for large scale real-time systems”. In: *Innovations in Systems and Software Engineering* 3.1 (2007), pp. 33–52. ISSN: 1614-5054. DOI: 10.1007/s11334-006-0015-7. URL: http://dx.doi.org/10.1007/s11334-006-0015-7 (cit. on p. 54).

[9] O. Finkel. “Undecidable Problems About Timed Automata”. In: *CoRR* abs/0712.1363 (2007). URL: http://arxiv.org/abs/0712.1363 (cit. on pp. iii, 3, 29).

[10] S. Guha, C. Narayan, and S. Arun-Kumar. “Reducing Clocks in Timed Automata while Preserving Bisimulation”. In: *CONCUR 2014 – Concurrency Theory: 25th International Conference, CONCUR 2014, Rome, Italy, September 2-5, 2014. Proceedings*. Ed. by P. Baldan and D. Gorla. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 527–543. ISBN: 978-3-662-44584-6. DOI: 10.1007/978-3-662-44584-6_36. URL: http://dx.doi.org/10.1007/978-3-662-44584-6_36 (cit. on p. iii).

[11] K. G. Larsen, P. Pettersson, and W. Yi. *UPPAAL in a Nutshell*. 1997 (cit. on p. 12).

[12] T. Lengauer and R. E. Tarjan. “A Fast Algorithm for Finding Dominators in a Flowgraph”. In: vol. 1. 1. New York, NY, USA: ACM, Jan. 1979, pp. 121–141. DOI: 10.

1145/357062.357071. URL: http://doi.acm.org/10.1145/357062.357071 (cit. on p. 15).

[13] N. Saeedloei. *From Scenarios to Timed Automata*. Technical Report. Duluth, MN: Department of Computer Science, University of Minnesota Duluth, June 2016. URL: http://www.d.umn.edu/cs/research/Neda%20Saeedloei%20Technical%20Report%2016-01.pdf (cit. on pp. 15, 21).

[14] N. Saeedloei and F. Kluzniak. *Optimal Clock Allocation for a Class of Timed Automata*. Technical Report. Duluth, MN: Department of Computer Science, University of Minnesota Duluth, Sept. 2016. URL: http://www.d.umn.edu/cs/research/documents/technicalReport-Neda-3.pdf (cit. on p. 28).

[15] S. Somé, R. Dssouli, and J. Vaucher. "From Scenarios to Timed Automata: Building Specifications from Users Requirements". In: *2nd Asia-Pacific Software Engineering Conference (APSEC '95), December 6-9, 1995, Brisbane, Queensland, Australia*. 1995, pp. 48–57. DOI: 10.1109/APSEC.1995.496953. URL: http://dx.doi.org/10.1109/APSEC.1995.496953 (cit. on p. 2).

[16] S. Uchitel, G. Brunet, and M. Chechik. "Synthesis of Partial Behavior Models from Properties and Scenarios". In: *IEEE Trans. Software Eng.* 35.3 (2009), pp. 384–406. DOI: 10.1109/TSE.2008.107. URL: http://dx.doi.org/10.1109/TSE.2008.107 (cit. on p. 2).

[17] S. Uchitel, J. Kramer, and J. Magee. "Synthesis of Behavioral Models from Scenarios." In: *IEEE Trans. Software Eng.* 29.2 (2003), pp. 99–115. URL: http://dblp.uni-trier.de/db/journals/tse/tse29.html#UchitelKM03 (cit. on p. 2).

[18] S. Yovine. "A case study: the CSMA/CD protocol". In: (Nov. 1994) (cit. on p. 55).